Department of Computer Science
Series of Publications A
Report A-2015-3

Efficient Construction of Fundamental Data Structures in Large-Scale Text Indexing

Dominik Kempa

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Linus Torvalds Auditorium (B123), Exactum, Gustaf Hällströmin katu 2b, on October 2nd, 2015, at 10 o'clock in the morning.

University of Helsinki
Finland

Supervisors
Juha Kärkkäinen, University of Helsinki, Finland
Esko Ukkonen, University of Helsinki, Finland

Pre-examiners
Johannes Fischer, Technische Universität Dortmund, Germany
Jorma Tarhio, Aalto University, Finland

Opponent
Gonzalo Navarro, University of Chile, Chile

Custos
Esko Ukkonen, University of Helsinki, Finland

Contact information

Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2b)
FI-00014 University of Helsinki
Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/
Telephone: +358 2941 911, telefax: +358 9 876 4314

ISSN 1238-8645
ISBN 978-951-51-1535-5 (paperback)
ISBN 978-951-51-1536-2 (PDF)
Computing Reviews (1998) Classification: E.1, E.4, F.2.2
Helsinki 2015
Unigrafia

Efficient Construction of Fundamental Data Structures in Large-Scale Text Indexing

Dominik Kempa

Department of Computer Science
P.O. Box 68, FI-00014 University of Helsinki, Finland
dominik.kempa@cs.helsinki.fi
http://www.cs.helsinki.fi/dominik.kempa/

PhD Thesis, Series of Publications A, Report A-2015-3
Helsinki, September 2015, 68 + 88 pages
ISSN 1238-8645
ISBN 978-951-51-1535-5 (paperback)
ISBN 978-951-51-1536-2 (PDF)

Abstract

This thesis studies efficient algorithms for constructing the most fundamental data structures used as building blocks in (compressed) full-text indexes. Full-text indexes are data structures that allow efficiently searching for occurrences of a query string in a (much larger) text. We are mostly interested in large-scale indexing, that is, dealing with input instances that cannot be processed entirely in internal memory and thus a much slower, external memory needs to be used. Specifically, we focus on three data structures: the suffix array, the LCP array and the Lempel-Ziv (LZ77) parsing. These are routinely found as components or used as auxiliary data structures in the construction of many modern full-text indexes.

The suffix array is a list of all suffixes of a text in lexicographical order. Despite its simplicity, the suffix array is a powerful tool used extensively not only in indexing but also in data compression, string combinatorics or computational biology. The first contribution of this thesis is an improved algorithm for external memory suffix array construction based on constructing suffix arrays for blocks of text and merging them into the full suffix array.

In many applications, the suffix array needs to be augmented with the information about the longest common prefix between each two adjacent suffixes in lexicographical order. The array containing such information is

called the longest-common-prefix (LCP) array. The second contribution of this thesis is the first algorithm for computing the LCP array in external memory that is not an extension of a suffix-sorting algorithm.

When the input text is highly repetitive, the general-purpose text indexes are usually outperformed (particularly in space usage) by specialized indexes. One of the most popular families of such indexes is based on the Lempel-Ziv (LZ77) parsing. LZ77 parsing is the encoding of text that replaces long repeating substrings with references to other occurrences. In addition to indexing, LZ77 is a heavily used tool in data compression. The third contribution of this thesis is a series of new algorithms to compute the LZ77 parsing, both in RAM and in external memory.

The algorithms introduced in this thesis significantly improve upon the prior art. For example: (i) our new approach for constructing the LCP array in external memory is faster than the previously best algorithm by a factor of 2–4 and simultaneously reduces the disk space usage by a factor of four; (ii) a parallel version of our improved suffix array construction algorithm is able to handle inputs much larger than considered in the literature so far. In our experiments, computing the suffix array of a 1 TiB file with the new algorithm took a little over a week and required only 7.2 TiB of disk space (including input and output), whereas on the same machine the previously best algorithm would require 3.5 times as much disk space and take about four times longer.

Computing Reviews (1998) Categories and Subject Descriptors:
E.1 [Data Structures]: String data structures
E.4 [Coding and Information Theory]: Data compaction and compression - *Lempel-Ziv compression, Burrows-Wheeler transform*
F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems - *text indexing*

General Terms:
algorithms, experimentation, performance

Additional Key Words and Phrases:
external memory algorithms, algorithm engineering, full-text indexes, suffix array, LCP array, Burrows-Wheeler transform, Lempel-Ziv factorization, Lempel-Ziv parsing, LZ77, string processing, data compression

Acknowledgements

First, I would like to thank my supervisors, Juha Kärkkäinen and Esko Ukkonen, for their excellent guidance, dedication and support throughout my entire PhD studies. I would also like to thank Simon Puglisi for being an exceptional mentor in numerous aspects of scientific and everyday life. I am grateful to pre-examiners Johannes Fischer and Jorma Tarhio for their feedback, and to Marina Kurtén for improving the language of this thesis.

During my studies I was lucky to find myself surrounded by a number of talented researchers in the field of string algorithms and bioinformatics, including Djamal Belazzougui, Fabio Cunial, Travis Gagie, Pekka Mikkola, Veli Mäkinen, Leena Salmela, Jouni Sirén, Alexandru Tomescu, Daniel Valenzuela, and Niko Välimäki. I wish to thank you all for inspiring discussions and for enjoyable company outside of working hours.

I would like to acknowledge the efficient organization and friendly atmosphere of the Department of Computer Science at the University of Helsinki that has provided me with excellent working conditions, administrative support, and IT infrastructure. I also want to thank the IT staff, in particular Ville Hautakangas, Jani Jaakkola, and Pekka Niklander for excellent IT services.

This research has been supported by the Department of Computer Science at the University of Helsinki, the Finnish Centre of Excellence for Algorithmic Data Analysis Research (ALGODAN), the Doctoral Programme in Computer Science (DoCS), and the Helsinki Institute for Information Technology (HIIT).

Lastly, and most importantly, I would like to thank my parents, Maria and Tadeusz Kempa, for their encouragement and unconditional support.

Helsinki, 6th of September, 2015
Dominik Kempa

Original Papers

The thesis consists of a summarizing overview and the following five peer-reviewed publications, referred to as Paper I–V. These publications are reproduced at the end of the thesis. None of the publications have been used in previous dissertations.

I. Juha Kärkkäinen and Dominik Kempa. **Engineering a Lightweight External Memory Suffix Array Construction Algorithm**. Accepted for publication in *Mathematics in Computer Science*.

II. Juha Kärkkäinen and Dominik Kempa. **LCP Array Construction in External Memory**. Extended version of the paper accepted to the 13th Symposium on Experimental Algorithms (SEA 2014), LNCS 8504, pages 412–423, 2014. Submitted to *ACM Journal of Experimental Algorithmics* (special issue of SEA 2014).

III. Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. **Lazy Lempel-Ziv Factorization Algorithms**. Accepted for publication in *ACM Journal of Experimental Algorithmics* (special issue of the 2013 Workshop on Algorithm Engineering and Experiments (ALENEX 2013)).

IV. Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. **Lightweight Lempel-Ziv Parsing**. 12th Symposium on Experimental Algorithms (SEA 2013), LNCS 7933, pages 139–150, 2013.

V. Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. **Lempel-Ziv Parsing in External Memory**. 2014 Data Compression Conference (DCC 2014), pages 153–162, 2014.

Implementations of all algorithms presented in the papers are available at `http://www.cs.helsinki.fi/group/pads/`.

Contents

Chapter 1

Introduction

1.1 Motivation

This thesis is concerned with the problems of efficiently indexing large collections of textual data. Sources of such data include multi-author databases of documents, such as Wikipedia [75], collections of versioned source code, and web crawls [24]. More recently, with the advent of high-throughput DNA-sequencing machines [20, 33], enormous volumes of genomic data that requires indexing have been released. These databases keep growing in size. A plan of sequencing genomes of up to one hundred thousand citizens was recently announced by the British government [1]. This project will produce half a petabyte of data.

The field of *compressed full-text indexing*, which combines aspects of information theory, data compression, and combinatorial pattern matching, aims to address the problem of storing and searching such data. Compressed full-text indexes are data structures that require small space (close to the data in compressed form) but that simultaneously support various types of queries over the underlying data (searching, for example) without decompressing the data. The area has been witness to intense research in the past decade [63], and several of its fruits are now widely used in molecular biology [75].

With the drastic increase in the size of data requiring indexing, scalable construction of these indexes has emerged as a pressing open problem. While the resulting compressed indexes are often small enough to fit in RAM [55], the input data and intermediate data structures used for building the index are too big for RAM and require efficient use of much slower external memory [22].

The aim of this thesis is to address the problem of efficient construction of full-text indexes for very large inputs. Rather than optimizing a construction of specific index, we focus on the construction of the basic data structures that are building blocks of many existing indexes. Aside from indexing, these data structures also have many other applications in string processing.

Our methodology is best described as *algorithm engineering* [72]. In a classical methodology, the algorithm is usually first designed to achieve good time/space complexity in a particular theoretical model. The implementation is then often performed by practitioners, i.e., the result is a combined effort of at least two parties.

In contrast, in algorithm engineering, the design, analysis, implementation, and experimental evaluation of the algorithm is the cycle driving algorithmic research. All phases form a feedback loop that cycles through until the final design emerges. This way, the discrepancy between theoretical models and actual hardware (e.g., cache misses) is greatly reduced. As a result, the implementation shares some design load and vice-versa.

- The first data structure studied in this thesis is the *suffix array*, a lexicographically sorted list of suffixes of text. It was introduced [58, 36] as a space-efficient alternative to suffix trees [79]. The suffix array is a simple, yet powerful data structure. It provides efficient and often optimal solutions to many problems involving pattern matching and pattern discovery on large data sets, arising in practically important domains such as biological sequence analysis [65].

  Being space-efficient, the suffix array is widely used in practice. For example, it is vital in construction of text indexes, such as FM-index [23], which relies on the existence of suffix array samples. Furthermore, the main component of FM-index is the *Burrows-Wheeler transform (BWT)* [11] – a transformation of input text that can either be obtained from the suffix array or by modifying the suffix array construction algorithm to produce BWT instead of or in addition to the suffix array. Either way, the advancements in suffix array construction automatically translate to faster index construction.

- The second data structure studied in this thesis is the *longest-common-prefix (LCP) array* [58]. It stores the lengths of common prefixes between adjacent suffixes in the suffix array. The LCP array is an essential component of many text indexes, including both traditional indexes such as enhanced suffix array [3] and compressed variants, such as compressed suffix tree [34, 66].

The LCP array is also the most commonly used addition to the suffix array. Modern textbooks spend dozens of pages describing applications, where the suffix array needs to be augmented with the LCP array to achieve optimal time complexity [65].

- The third data structure we consider is the *Lempel-Ziv factorization* [82] (also known as *LZ77 parsing*). It is a partition of the text into substrings (called phrases), such that each phrase has another occurrence to the left and the total number of phrases is minimized. LZ77 parsing (or a grammar derived from LZ77 parsing) is a basic ingredient of many text indexes designed for storing and searching highly repetitive sequences [55, 32, 30, 21, 31].

  LZ77 also has numerous other applications outside indexing, e.g., efficient detection of repetitions in strings [51, 6]. It is also the computational bottleneck in many popular lossless compressors, such as gzip or 7zip [70] (the latter being one of the most powerful general-purpose compressors).

1.2 Original papers and contributions

Below we give a brief summary of each of the papers used in the thesis together with the author's contribution.

Paper I. We describe a novel algorithm for constructing the suffix array in external memory. The basic idea of the algorithm goes back 20 years [15], but we describe several new improvements that make the algorithm much faster. The resulting algorithm is the fastest suffix array construction algorithm when the size of the text is within a factor of about five from the size of the RAM in either direction (which is a common situation in practice), and uses about three times less disk space than competitors.

I was responsible for implementation, experiments and writing the corresponding section of the paper.

Paper II. We give the first external memory algorithm for constructing the LCP array from the suffix array of the input text. The only previously known way to construct the LCP array in external memory was to modify a suffix array construction algorithm to output the LCP array as a by-product. Compared to previous methods, our approach results in an algorithm that is about three times faster and uses about a quarter of the disk space.

I co-designed and implemented the algorithm, performed the experiments, and co-wrote the paper.

Paper III. In this paper we give several new algorithms for constructing the LZ77 parsing in internal memory. The new algorithms consistently outperform all previous methods in practice, both in runtime and space usage. All the studied algorithms are "large space" in the sense that they require $\Theta(n \log n)$ bits of working space for a string of length n.

I was involved in the design and analysis of the new algorithms and implemented most of the algorithms.

Paper IV. This paper introduces a new algorithm for computing the LZ77 parsing in $\mathcal{O}(dn)$ time and using $\mathcal{O}((n \log n)/d)$ bits of working space for any $d \geq 1$, thus addressing the shortcoming of algorithms in Paper III. The algorithm still operates in internal memory, but allows reducing the working space to little over what is necessary to hold the text.

My main contribution was implementing the algorithms and carrying out the experiments. I participated in the design of the new algorithm for matching statistics inversion.

Paper V. In this paper we study the problem of computing LZ77 parsing, when the text (greatly) exceeds the amount of available RAM, thus making even the algorithm from Paper IV unusable. We propose three new algorithms and show that each of the algorithms has its niche (a combination of input and system parameters), where it is the best of all algorithms.

Design of the algorithms was a joint work. I also implemented the algorithms and wrote the experimental section of the paper.

1.3 Outline

The present overview summarizes the original papers I–V and is organized as follows. Chapter 2 of the overview contains preliminary definitions and the computational model used to analyze the algorithms.

Chapter 3 describes the new external-memory suffix array construction algorithm. The text is based on Paper I. At the end of the chapter we include a discussion about the possible improvements and our follow-up work on the algorithm, namely, implementing the algorithm in the multi-core architecture.

Chapter 4 discusses the external-memory LCP array construction and includes the new algorithm from Paper II. The chapter takes an approach

at the algorithm exposition that differs from the original paper. We take an internal-memory algorithm and gradually transform it into external-memory algorithm, explaining intuitions and design decisions on the way.

Chapter 5 contains a description of several new algorithms computing the LZ77 factorization in internal memory, as well as an overview of existing approaches. Several of the concepts and methods introduced in the chapter are then developed into fully external memory algorithms in the next chapter. The material in this chapter is based on Papers III and IV.

In Chapter 6 we discuss the computation of LZ77 parsing in external-memory. The chapter includes new algorithms from Paper V, some of which require the suffix and LCP array as an input. Thus, the chapter also motivates and consolidates the findings from previous chapters. As an extension to Paper V, we describe a modification to the most scalable of the algorithms (EM-LPF) that significantly reduces the peak disk space usage and present its experimental evaluation. We also revised some experiments from Paper V to include the new algorithms developed since its publication.

The overview ends with a brief summary and discussion on future work in Chapter 7.

Chapter 2

Preliminaries

2.1 Strings

Let $\Sigma = \{0, 1, \ldots, \sigma - 1\}$ be an *alphabet* of size σ. The elements of Σ are called *letters* or *characters*. A finite sequence of letters is called a *string* or *text*. The length of string S is denoted $|S|$ and the $(i+1)^{\text{th}}$ letter of string S is denoted by $S[i]$, that is, for $|S| = n$ the elements of S are $S[0]$, $S[1], \ldots, S[n-1]$.

A subsequence $S[i], \ldots, S[j]$ for $0 \leq i \leq j < n$ is called a *substring* of S and is denoted as $S[i..j]$. We write $S[i..j)$, $0 \leq i \leq j \leq n$ as a shorthand for $S[i..j-1]$. When $i = j$, $S[i..j)$ is an *empty string*, also denoted by ε. A substring $S[i..j)$ of S is called a *prefix* of S if $i = 0$ and a *suffix* of S if $j = n$. A prefix/substring/suffix S' of S is called *proper* if $S' \neq S$. A *concatenation* of two strings S and S' is $SS' = S[0] \ldots S[|S|-1]S'[0] \ldots S'[|S'|-1]$.

We define the *lexicographical order* between the strings as follows. For any S and S' we have $S < S'$ iff S is a proper prefix of S' or there exists $i < \min(|S|, |S'|)$ such that $S[0..i) = S'[0..i)$ and $S[i] < S'[i]$.

2.2 Full-text indexes

Suffix array and Burrows-Wheeler transform. Let T be a text of length n. The *suffix array (SA)* of T is an array $\mathsf{SA}_\mathsf{T}[0..n)$ containing a permutation of integers $\{0, \ldots, n-1\}$ such that $\mathsf{T}[\mathsf{SA}_\mathsf{T}[0]..n) < \mathsf{T}[\mathsf{SA}_\mathsf{T}[1]..n) < \ldots < \mathsf{T}[\mathsf{SA}_\mathsf{T}[n-1]..n)$. Whenever T is clear from the context we drop the subscript and just write SA.

The *inverse suffix array (ISA)* of T is defined as an array $\mathsf{ISA}[0..n)$ such that $\mathsf{ISA}[\mathsf{SA}[i]] = i$. In other words, $\mathsf{ISA}[j]$ is the rank (number of smaller

i	$\mathsf{SA}[i]$	$\mathsf{LCP}[i]$	$\mathsf{BWT}[i]$	$\mathsf{T}[\mathsf{SA}[i]..\lvert\mathsf{T}\rvert)$
0	3	0	b	aabbabbab
1	10	1	b	ab
2	1	2	b	abaabbabbab
3	7	2	b	abbab
4	4	5	a	abbabbab
5	11	0	a	b
6	2	1	a	baabbabbab
7	9	2	b	bab
8	0	3	$	babaabbabbab
9	6	3	b	babbab
10	8	1	a	bbab
11	5	4	a	bbabbab

Table 2.1: Suffix array, LCP array and Burrows-Wheeler transform for text $\mathsf{T} = \texttt{babaabbabbab}$.

suffixes) of suffix j in ascending lexicographical order of all suffixes of T. [1]

The *Burrows-Wheeler transform (BWT)* of T is an array $\mathsf{BWT}[0..n)$ such that $\mathsf{BWT}[i] = \mathsf{T}[\mathsf{SA}[i]-1]$ if $\mathsf{SA}[i] > 0$ and $\mathsf{BWT}[i] = \$$ otherwise, where $\$ \notin \Sigma$ is a special symbol smaller than all other symbols in the alphabet.

LCP array. The *LCP array* of a text T is defined as an array $\mathsf{LCP}[0..n)$, such that $\mathsf{LCP}[i] = \mathsf{lcp}(\mathsf{SA}[i], \mathsf{SA}[i-1])$, $i \in [1..n)$ and $\mathsf{LCP}[0] = 0$, where $\mathsf{lcp}(i_1, i_2)$ denotes the length of the longest common prefix of suffix i_1 and i_2 of T. An example of suffix array, LCP array and BWT is given in Table 2.1.

LZ77. For a string T, the *longest previous factor (LPF)* at position j, denoted $\mathsf{LPF}_\mathsf{T}[j]$, is a pair (p_j, ℓ_j) such that, $p_j < j$, $\mathsf{T}[p_j..p_j+\ell_j) = \mathsf{T}[j..j+\ell_j)$ and $\ell_j > 0$ is maximized. When $\mathsf{T}[j]$ is the leftmost occurrence of a symbol, we define $p_j = \mathsf{T}[j]$ and $\ell_j = 0$. In other words, $\mathsf{T}[j..j+\ell_j)$ is the longest prefix of $\mathsf{T}[j..n)$ that also occurs at some position $p_j < j$ in T. Note that p_j is not unique but ℓ_i is.

The *LZ77 factorization* (or *LZ77 parsing*) of T is a partition $\mathsf{T} = f_1 f_2 \cdots f_z$ defined recursively as follows. Assume that $f_1 \cdots f_{i-1}$ is the LZ77 parsing of $\mathsf{T}[0..j)$. Then the next element of the parsing is $f_i = \mathsf{T}[j..j+\ell)$, where $\ell = \max\{1, \ell_j\}$. The substring f_i for $i = 1, \ldots, z$ is called an *LZ-factor* or *LZ-phrase*. Each LZ-factor f_i is encoded as $\mathsf{LPF}[j]$, where $j = |f_1 \cdots f_{i-1}|$. If $\ell_j > 0$, then the substring $\mathsf{T}[p_j..p_j+\ell_j)$ is called the

[1] Whenever possible, i is used to denote the positions in the lexicographical order, and j to denote the positions in the text order, e.g., we write $\mathsf{SA}[i]$, $\mathsf{ISA}[j]$.

i	LPF p_i	ℓ_i	0 1 2 3 4 5 6 7 8 9 10 11
			(b)(a)(b)(b a b)(a b b)(b a b)
0	b	0	b a b b a b a b b b a b
1	a	0	b a b b a b a b b b a b
2	0	1	(b a (b))b a b a b b b a b
3	0	3	(b a b)(b a b)a b b b a b
4	1	2	b(a b)b(a b)a b b b a b
5	0	4	(b a b b)a(b a b b)b a b
6	1	3	b(a b b)a b(a b b)b a b
7	2	2	b a(b b)a b a(b b)b a b
8	2	4	b a(b b a b)a b(b b a b)
9	3	3	b a b(b a b)a b b(b a b)
10	4	2	b a b b(a b)a b b b(a b)
11	5	1	b a b b a(b)a b b b a(b)

LZ77: (b,0),(a,0),(0,1),(0,3),(1,3),(3,3)

Figure 2.1: Example of the LPF array and LZ77 parsing for text T = babbababbbab.

source of the LZ-factor f_i. The example of LPF pairs and LZ77 parsing is given in Figure 2.1.

2.3 Computational model

For the analysis of internal-memory algorithms we assume the standard RAM model, where all elementary arithmetic operations on $\log n$-bit words (including multiplication) take constant time.

To analyze the external memory algorithms we use the standard External Memory (EM) model [78]. In this model, the memory in the system consists of a fast random-access memory of size M (measured in $\log n$-bit words) and slow secondary memory (disk) of unbounded size. The CPU can at any point access any cell in RAM, but not on disk. Any access to data on disk requires transferring the whole block of $B \log n$-bit words from disk to RAM first. Similarly, writing data to disk is also performed in blocks. The performance measures of an algorithm in this model are:

- time complexity: the number of operations performed by the CPU, measured as in the standard RAM model,
- I/O complexity: the number of blocks moved between disk and RAM. A single transfer (read of write) of a block is called an *I/O*,

- peak disk space usage: the maximum number of disk blocks used by the algorithm at any time, over the course of the whole algorithm.

Throughout this thesis we use the following fundamental result.

Theorem 2.1 ([4])**.** *The I/O complexity of sorting n (tuples of) integers in the EM model is $\Theta((n/B)\log_{M/B}(n/B))$.*

We will often write $\mathcal{O}(\mathrm{sort}(n))$ as a shorthand for $\mathcal{O}((n/B)\log_{M/B}(n/B))$.

To simplify the analysis, in this thesis we make the following weak assumptions about the basic model parameters: $M = \mathcal{O}(n)$ (where n is the length of the input text measured in characters), $M = \Omega(\log n)$ and $B = \mathcal{O}\big(M^{1-\epsilon}\big)$ for some constant $\epsilon > 0$.

Chapter 3

Suffix array construction

The suffix array [36, 58] of a string, a lexicographically sorted list of all its suffixes, is one of the most important data structures in string processing. It serves as a prerequisite to many efficient string algorithms [65] and plays a central role in text indexing [63].

Over the years, many internal-memory suffix sorting algorithms have been proposed. Some of those algorithms achieve the optimal $\mathcal{O}(n)$ time complexity [46, 50, 64], use little or almost no extra working space and are very fast in practice [61, 60]. All those algorithms, however, require $\Omega(n \log n)$ bits of space, and thus can only be used for inputs that are a few times smaller than the size of RAM. For many real world inputs such as Wikipedia or DNA databases this is not enough.

To escape the limitations of RAM, external-memory algorithms for suffix array construction have been proposed, some of which have theoretically optimal I/O complexity $\mathcal{O}((n/B) \log_{M/B}(n/B))$ and time complexity $\mathcal{O}(n \log_{M/B}(n/B))$ [47, 10]. These algorithms can operate within a moderate RAM budget. The biggest obstacle in their use is, however, a substantial disk space usage. The currently fastest implementation with optimal I/O complexity, eSAIS, requires $28n$ bytes of disk space for input of n bytes, assuming each integer is represented using 40 bits.

In this chapter we describe a new external-memory algorithm for suffix array construction called SAscan. The algorithm has $\Omega(n^2/M)$ time complexity, thus for large values of n/M it is outperformed by eSAIS. However, when the input size n is within some small factor from the RAM size M (which is a common situation in practice), SAscan is the fastest suffix-sorting algorithm. The algorithm is also very space efficient, it requires very little disk space in addition to what is necessary to store input text and output suffix array. The algorithm is based on the BWT construction

i	$\mathsf{gap}_{\mathsf{X:Y}}[i]$	$\mathsf{SA}_{\mathsf{X:Y}}[i]$	$\mathsf{T}[\mathsf{SA}_{\mathsf{X:Y}}[i]..\lvert\mathsf{T}\rvert)$
0	0	3	`aabbabbab`
1	1	1	`abaabbabbab`
2	3	2	`baabbabbab`
3	1	0	`babaabbabbab`
4	3		

i	$\mathsf{SA}_{\mathsf{T}}[i]$	$\mathsf{T}[\mathsf{SA}_{\mathsf{T}}[i]..\lvert\mathsf{T}\rvert)$
0	3	`aabbabbab`
1	10	`ab`
2	1	`abaabbabbab`
3	7	`abbab`
4	4	`abbabbab`
5	11	`b`
6	2	`baabbabbab`
7	9	`bab`
8	0	`babaabbabbab`
9	6	`babbab`
10	8	`bbab`
11	5	`bbabbab`

Figure 3.1: Example of the partial suffix array and gap array. Input string are T = `babaabbabbab`, $\mathsf{X} = \mathsf{T}[0..4)$ = `baba`, and $\mathsf{Y} = \mathsf{T}[4..\lvert\mathsf{T}\rvert)$ = `abbabbab`. Observe that the sequence of suffixes in $\mathsf{SA}_{\mathsf{X:Y}}$ (left) is a subsequence of $\mathsf{SA}_{\mathsf{XY}} = \mathsf{SA}_{\mathsf{T}}$ (right).

algorithm of Ferragina et al. [22]. [1] Our description of the algorithm loosely follows the original, but we highlight the changes and improvements that we made. The material is based on Paper I.

3.1 Preliminaries

To represent a lexicographical ordering of a subset of suffixes we first introduce a slight variation of a suffix array. Let X, Y be strings and let $m = |\mathsf{X}|$. A *partial suffix array* $\mathsf{SA}_{\mathsf{X:Y}}$ is an array $\mathsf{SA}_{\mathsf{X:Y}}[0..m)$ containing a permutation of integers $\{0, \ldots, m-1\}$ such that $\mathsf{X}[\mathsf{SA}_{\mathsf{X:Y}}[0]..m)\mathsf{Y} < \mathsf{X}[\mathsf{SA}_{\mathsf{X:Y}}[1]..m)\mathsf{Y} < \ldots < \mathsf{X}[\mathsf{SA}_{\mathsf{X:Y}}[m-1]..m)\mathsf{Y}$.

Partial suffix arrays play a central role in the new algorithm introduced in this chapter. The key property of $\mathsf{SA}_{\mathsf{X:Y}}$ is that it contains all suffixes of XY with the starting position in X in exactly the same order as the full suffix array $\mathsf{SA}_{\mathsf{XY}}$. Therefore $\mathsf{SA}_{\mathsf{XY}}$ can be obtained by interleaving the elements of $\mathsf{SA}_{\mathsf{X:Y}}$ and SA_{Y}. The example of partial suffix array is given in Figure 3.1.

[1] Although originally designed to construct BWT, the algorithm of Ferragina et al. can be easily modified to compute the suffix array instead of or in addition to BWT. Analogously, our algorithm can be modified to compute the BWT.

3.2 Algorithm description

3.2.1 Overview

Let $\mathsf{T}[0..n)$ be the input text and let m be a positive integer. The algorithm divides the input text into $d = \lceil n/m \rceil$ *segments*[2], each of size (at most) m. The segments are processed right-to-left. Each of the segments is processed in RAM, thus m is chosen so that all data structures built during the segment processing can fit in RAM. During the segment processing we also use disk space, but all accesses are purely sequential.

Denote the currently processed segment by X and let Y be the concatenation of the segments to the right of X. For simplicity assume that the size of X is exactly m. Suppose that SA_Y was already computed and is stored on disk. The goal is to have SA_XY when the processing of the segment X is finished.

To accomplish this, we first compute the partial suffix array $\mathsf{SA}_{\mathsf{X:Y}}[0..m)$. The details of this step are described in Section 3.2.2. For now, we observe that the suffixes in $\mathsf{SA}_{\mathsf{X:Y}}$ are in the same relative order as the suffixes in SA_XY, thus to merge $\mathsf{SA}_{\mathsf{X:Y}}$ with SA_Y to obtain SA_XY we need to know how many suffixes of Y should be placed between any pair of lexicographically adjacent suffixes of XY starting in X. This information is computed and stored in the form of *gap array* $\mathsf{gap}_{\mathsf{X:Y}}[0..m]$ which is formally defined as follows. Let $s_i = \mathsf{SA}_{\mathsf{X:Y}}[i]$. For $i \in (0..m)$,

$$\mathsf{gap}_{\mathsf{X:Y}}[i] = |\{s \in [0..|\mathsf{Y}|) : \mathsf{X}[s_{i-1}..m)\mathsf{Y} < \mathsf{Y}[s..|\mathsf{Y}|) < \mathsf{X}[s_i..m)\mathsf{Y}\}|.$$

The remaining values are defined as:

$$\mathsf{gap}_{\mathsf{X:Y}}[0] = |\{s \in [0..|\mathsf{Y}|) : \mathsf{Y}[s..|\mathsf{Y}|) < \mathsf{X}[s_0..m)\mathsf{Y}\}|,$$
$$\mathsf{gap}_{\mathsf{X:Y}}[m] = |\{s \in [0..|\mathsf{Y}|) : \mathsf{X}[s_{m-1}..m)\mathsf{Y} < \mathsf{Y}[s..|\mathsf{Y}|)\}|.$$

The example of gap array is given in Figure 3.1. The details of $\mathsf{gap}_{\mathsf{X:Y}}$ array construction are described in Section 3.2.3. Once the gap array has been computed, we can easily merge $\mathsf{SA}_{\mathsf{X:Y}}$ with SA_Y to obtain SA_XY. Doing this for every segment would introduce a lot of I/O, thus we approach the merging differently. At the end of segment processing, we write $\mathsf{SA}_{\mathsf{X:Y}}$ and $\mathsf{gap}_{\mathsf{X:Y}}$ to disk instead. Once all segments have been processed, we perform a combined merging of all partial suffix arrays into the final suffix array. The details of this step are described in Section 3.2.4.

[2]Unlike in Paper I, we use the term *segment*, rather than *block* to denote the contiguous sequence of characters in the text. This is to unify the notation in the thesis and to avoid the confusion with disk blocks (which are basic units of I/O in the EM model).

Handling long suffix comparison. During construction of $\mathsf{SA}_{\mathsf{X:Y}}$ we need to compare suffixes of XY starting in X. This may require $\Omega(|\mathsf{Y}|)$ symbol comparisons if we perform it naively, and since we can have $|\mathsf{Y}| \gg |\mathsf{X}|$, it would introduce a lot of I/O.

To speed up comparisons of suffixes with long common prefix we introduce the gt_{Y} bitvector. Formally, for $i \in [0..|\mathsf{Y}|)$,

$$\mathsf{gt}_{\mathsf{Y}}[i] = \begin{cases} 1 & : \text{ if } \mathsf{Y}[i..|\mathsf{Y}|) > \mathsf{Y} \\ 0 & : \text{ if } \mathsf{Y}[i..|\mathsf{Y}|) \leq \mathsf{Y} \end{cases}.$$

With gt_{Y}, comparing two suffixes of XY starting in X requires accessing at most m symbols of Y. This is formalized as follows.

Observation 3.1. *For $0 \leq i < j < m$ the comparison*

$$\mathsf{X}[i..m)\mathsf{Y} < \mathsf{X}[j..m)\mathsf{Y}$$

is equivalent to

$$\mathsf{X}[i..m) < \mathsf{X}[j..m)\mathsf{Y}[0..j-i) \text{ or } \mathsf{X}[i..m) = \mathsf{X}[j..m)\mathsf{Y}[0..j-i) \text{ and } \mathsf{gt}_{\mathsf{Y}}[j-i] = 1.$$

From now we assume that when we process segment X, the bitvector gt_{Y} is available on disk and that in addition to $\mathsf{SA}_{\mathsf{X:Y}}$ and $\mathsf{gap}_{\mathsf{X:Y}}$ we also produce $\mathsf{gt}_{\mathsf{XY}}$ as an output (to use for the next segment).

3.2.2 Constructing partial suffix array

We now have a closer look at the construction of the partial suffix array. Assume that X is the current segment under consideration and Y is the concatenation of the segments to the right of X. The goal is to obtain $\mathsf{SA}_{\mathsf{X:Y}}$.

The main idea is to compute a new string X' with the property $\mathsf{SA}_{\mathsf{X}'} = \mathsf{SA}_{\mathsf{X:Y}}$. This approach allows using a highly optimized existing implementation for suffix sorting and replacing it with a faster algorithm in the future, if such becomes available. The string X' is defined as:

$$\mathsf{X}'[i] = \begin{cases} \mathsf{X}[i] & : \text{ if } \mathsf{X}[i..m)\mathsf{Y} < \mathsf{X}[m-1]\mathsf{Y} \\ \mathsf{X}[i]+1 & : \text{ if } \mathsf{X}[i..m)\mathsf{Y} \geq \mathsf{X}[m-1]\mathsf{Y} \end{cases}.$$

The above construction is only possible if X does not contain a symbol with the maximal value in the alphabet $(\sigma - 1)$. If this is not the case, but there is at least one symbol $c \in \Sigma$ that does not occur in X, then we can decrease all symbols larger than c in X (which does not affect the ordering of suffixes) and thus free the maximal symbol. If none of the previous cases

hold, we cannot directly apply the transformation. However, this case too can be solved, though it may result in X' that is larger than X by a factor of $1 + (2/\sigma)$. The details of this construction are described in Paper I.

Lemma 3.2. $\mathsf{SA}_{\mathsf{X}'} = \mathsf{SA}_{\mathsf{X}:\mathsf{Y}}$.

Proof. Suppose the claim does not hold, i.e., there exists as least one pair of indices $i, j \in [0..m)$ such that lexicographical ordering between pairs of suffixes $\mathsf{X}[i..m)\mathsf{Y}$ and $\mathsf{X}[j..m)\mathsf{Y}$ is different from the ordering of $\mathsf{X}'[i..m)$ and $\mathsf{X}'[j..m)$. Without the loss of generality we assume $i < j$. Among all such pairs, let i, j be the pair with the largest j and assume $\mathsf{X}[i..m)\mathsf{Y} < \mathsf{X}[j..m)\mathsf{Y}$ and $\mathsf{X}'[i..m) > \mathsf{X}'[j..m)$ (the other case, $\mathsf{X}[i..m)\mathsf{Y} > \mathsf{X}[j..m)\mathsf{Y}$, $\mathsf{X}'[i..m) < \mathsf{X}'[j..m)$ is analogous).

The relation between suffixes implies that $\mathsf{X}[i] \leq \mathsf{X}[j]$ and $\mathsf{X}'[i] \geq \mathsf{X}'[j]$. However, from the definition of X' we have $\mathsf{X}'[i] \in \{\mathsf{X}[i], \mathsf{X}[i] + 1\}$ (and analogously for j) thus only four cases are possible:

1. $\mathsf{X}[i] + 1 = \mathsf{X}[j]$, $\mathsf{X}'[i] = \mathsf{X}[i] + 1$, $\mathsf{X}'[j] = \mathsf{X}[j]$,
2. $\mathsf{X}[i] = \mathsf{X}[j]$, $\mathsf{X}'[i] = \mathsf{X}[i] + 1$, $\mathsf{X}'[j] = \mathsf{X}[j]$,
3. $\mathsf{X}[i] = \mathsf{X}[j]$, $\mathsf{X}'[i] = \mathsf{X}[i]$, $\mathsf{X}'[j] = \mathsf{X}[j]$,
4. $\mathsf{X}[i] = \mathsf{X}[j]$, $\mathsf{X}'[i] = \mathsf{X}[i] + 1$, $\mathsf{X}'[j] = \mathsf{X}[j] + 1$.

Cases 1. and 2. are not possible, because the definition of X' would imply that $\mathsf{X}[i..m)\mathsf{Y} \geq \mathsf{X}[m-1]\mathsf{Y}$ and $\mathsf{X}[j..m)\mathsf{Y} < \mathsf{X}[m-1]\mathsf{Y}$ and thus $\mathsf{X}[j..m)\mathsf{Y} < \mathsf{X}[i..m)\mathsf{Y}$ which contradicts the initial assumption.

Thus, either 3. or 4. must hold. In both cases $\mathsf{X}[i] = \mathsf{X}[j]$ and $\mathsf{X}'[i] = \mathsf{X}'[j]$. From that we obtain that either $\mathsf{X}[i+1..m)\mathsf{Y} < \mathsf{X}[j+1..m)\mathsf{Y}$ and $\mathsf{X}'[i+1..m) > \mathsf{X}'[j+1..m)$ or $j = m-1$. Since we assumed that j was the largest with such property, it must be $j = m-1$. Then we obtain $\mathsf{X}[i..m)\mathsf{Y} < \mathsf{X}[j..m)\mathsf{Y} = \mathsf{X}[m-1]\mathsf{Y}$ thus from the definition of X' we have $\mathsf{X}'[i] = \mathsf{X}[i]$ and $\mathsf{X}'[j] = \mathsf{X}[j] + 1$ which contradicts both cases 3. and 4. □

The comparison $\mathsf{X}[i..m)\mathsf{Y} < \mathsf{X}[m-1]\mathsf{Y}$ can be performed for all $i \in [0..m)$ in $\mathcal{O}(m)$ time using a modified string matching algorithm [42]. We omit the details. Thus, computing $\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}$ takes $\mathcal{O}(m)$ time.

3.2.3 Constructing the gap array

In this section we describe how to compute the $\mathsf{gap}_{\mathsf{X}:\mathsf{Y}}$ array. The construction starts with segment X and $\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}$ in RAM. We first compute the *partial Burrows Wheeler transform* $\mathsf{BWT}_{\mathsf{X}:\mathsf{Y}}[0..m)$ defined as follows. For $i \in [0..m)$,

Algorithm `ComputeGap`
1: $r \leftarrow 0$
2: **for** $i \leftarrow |\mathsf{Y}| - 1$ **downto** 0 **do**
3: $\quad c \leftarrow \mathsf{Y}[i]$
4: $\quad r \leftarrow \mathsf{C}[c] + \mathsf{rank}_c(\mathsf{BWT}_{\mathsf{X:Y}}, r)$
5: $\quad$ **if** $c = \mathsf{X}[m-1]$ **and** $\mathsf{gt}_{\mathsf{Y}}[i+1] = 1$ **then** $r \leftarrow r + 1$
6: $\quad \mathsf{gap}_{\mathsf{X:Y}}[r] \leftarrow \mathsf{gap}_{\mathsf{X:Y}}[r] + 1$

Figure 3.2: Construction of $\mathsf{gap}_{\mathsf{X:Y}}$ array.

$$\mathsf{BWT}_{\mathsf{X:Y}}[i] = \begin{cases} \mathsf{X}[\mathsf{SA}_{\mathsf{X:Y}}[i] - 1] & : \text{if } \mathsf{SA}_{\mathsf{X:Y}}[i] > 0 \\ \$ & : \text{otherwise} \end{cases}.$$

We assume that $\$ \notin \Sigma$ is a special symbol smaller than all other symbols in the alphabet. $\mathsf{BWT}_{\mathsf{X:Y}}$ is easy to compute from X and $\mathsf{SA}_{\mathsf{X:Y}}$ in internal memory.

Once $\mathsf{BWT}_{\mathsf{X:Y}}$ is computed, we preprocess it for *rank* queries. A rank query $\mathsf{rank}_c(\mathsf{S}, i)$ for a string S, $c \in [0..\sigma)$ and $i \in [0..|\mathsf{S}|]$ is defined as the number of occurrences of symbol c in prefix $\mathsf{S}[0..i)$. A single rank query can be answered in $\mathcal{O}(\log(2 + (\log \sigma / \log\log n)))$ time using a data structure occupying $\mathcal{O}(m)$ space [7].

Finally, with a single scan of X, we compute the array $\mathsf{C}[0..\sigma)$, where

$$\mathsf{C}[c] = |\{i \in [0..m) : \mathsf{X}[i] < c\}|.$$

We are now ready to start the computation of the gap array. The pseudo-code of the algorithm is given in Figure 3.2. The algorithm is essentially the backward search procedure, used for pattern matching in compressed text indexes [23]. The correctness of the algorithm is preserved due to the following invariant that holds at the beginning of the for-loop:

- r is the number of suffixes of XY starting in X that are lexicographically smaller than $\mathsf{Y}[i+1..|\mathsf{Y}|)$.

The invariant is trivially satisfied for $i = |\mathsf{Y}| - 1$. To see that lines 4–5 preserve the invariant, note that for any suffix $\mathsf{X}[j..m)\mathsf{Y} < c\mathsf{Y}[i+1..m)$ we either have:

- $\mathsf{X}[j] < c$ (type I suffix),
- $j < m-1$, $\mathsf{X}[j] = c$, and $\mathsf{X}[j+1..m)\mathsf{Y} < \mathsf{Y}[i+1..m)$ (type II suffix),
- $j = m-1$, $\mathsf{X}[j] = c$, and $\mathsf{Y} < \mathsf{Y}[i+1..m)$ (type III suffix).

Algorithm `Merge`
1: **for** $k = 0$ **to** $\lceil n/m \rceil - 1$ **do** $i_k \leftarrow 0$
2: **for** $i = 0$ **to** $n - 1$ **do**
3: $k \leftarrow 0$
4: **while** $\mathsf{gap}_k[i_k] > 0$ **do**
5: $\mathsf{gap}_k[i_k] \leftarrow \mathsf{gap}_k[i_k] - 1$
6: $k \leftarrow k + 1$
7: $\mathsf{SA}_\mathsf{T}[i] \leftarrow \mathsf{SA}_k[i_k] + km$
8: $i_k \leftarrow i_k + 1$

Figure 3.3: Merging suffix arrays.

There is exactly $\mathsf{C}[c]$ suffixes of type I and $\mathsf{rank}_c(\mathsf{BWT}_{\mathsf{X:Y}}, r)$ suffixes of type II. The type III suffix is accounted for in line 5. The comparison $\mathsf{Y} < \mathsf{Y}[i+1..m)$ is implemented with equivalent check $\mathsf{gt}_\mathsf{Y}[i+1] = 1$.

All accesses to Y and gt_Y in `ComputeGap` are sequential, and thus can be efficiently implemented in external memory.

During this step we also compute gt_XY. To do this, we first scan $\mathsf{SA}_{\mathsf{X:Y}}$ to find the position i_XY such that $\mathsf{SA}_{\mathsf{X:Y}}[i_\mathsf{XY}] = 0$. During that scan we can easily compute $\mathsf{gt}_\mathsf{XY}[0..m)$: simply set $\mathsf{gt}_\mathsf{XY}[\mathsf{SA}_{\mathsf{X:Y}}[i]] = 1$ for all $i > i_\mathsf{XY}$. The remaining bits of gt_XY are computed in `ComputeGap` using the fact that $\mathsf{gt}_\mathsf{XY}[m+i] = 1$ iff $r_i > i_\mathsf{XY}$, where r_i is the value of r at the end of the for-loop processing $\mathsf{Y}[i]$.

3.2.4 Merging partial suffix arrays

The last step of the algorithm is merging the partial suffix arrays of the segments into the full suffix array of the text SA_T. The algorithm presented here is the first improvement of Paper I over the original algorithm of Ferragina et al. for BWT construction [22]. In the original algorithm, the partial BWT of the segment $\mathsf{BWT}_{\mathsf{X:Y}}$ was merged with BWT_Y immediately after segment processing. The new method delays the merging until the very end of the algorithm which substantially reduces the I/O volume and speeds up the algorithm.

For $k \in [0..\lceil n/m \rceil)$, by X_k denote the $(k+1)$-th text segment from the left and let $\mathsf{Y}_k = \mathsf{X}_{k+1} \cdots \mathsf{X}_{\lceil n/m \rceil - 1}$, $\mathsf{SA}_k = \mathsf{SA}_{\mathsf{X}_k:\mathsf{Y}_k}$, and $\mathsf{gap}_k = \mathsf{gap}_{\mathsf{X}_k:\mathsf{Y}_k}$.

The algorithm is given in Figure 3.3. A single step of the computation consists of finding the segment that contains the next suffix (among all suffixes of the text) in lexicographical order (lines 3–6), moving the suffix from the partial suffix array of that segment into the suffix array of the text

(line 7) and updating the auxiliary information (line 8). The correctness of the algorithm relies on the following invariant. At the beginning of the main for-loop, for any $k \in [0..\lceil n/m \rceil)$,

- i_k is the number of suffixes already moved from SA_k to SA_T,
- $\mathsf{gap}_k[i_k]$ is the number of suffixes remaining in $\mathsf{SA}_{k+1}, \ldots, \mathsf{SA}_{\lceil n/m \rceil - 1}$ that are smaller than $\mathsf{SA}_k[i_k]$.

The merging algorithm reads multiple files simultaneously, thus needs a small buffer of size B for each. We can afford this only if $n/m \leq M/B$. If this is not true, we need to perform multiway (M/B)-ary merge. Thus, in the general case the I/O complexity of merging is $\mathcal{O}((n/B)\log_{M/B}(n/B))$.

3.2.5 Theoretical analysis

We now analyze the complexity of the algorithm in the standard external memory (EM) model. All data structures required when processing a segment need $\mathcal{O}(m \log n)$ bits, thus we set $m = \Theta(M)$.

The time complexity is dominated by rank queries. We perform $\Theta(n^2/m)$ queries, and each takes $\mathcal{O}(\log(2 + (\log \sigma / \log\log n)))$ [7]. All other operations in the algorithm take $\mathcal{O}\big(n^2/m\big)$ time.

To construct all gap arrays, the algorithm reads $\mathcal{O}\big(n^2/m\big)$ characters from disk. Each character takes $\log \sigma$ bits and a single I/O transfers $B \log n$ bits, hence in total the gap array construction requires $\mathcal{O}\big(n^2/(MB\log_\sigma n)\big)$ I/Os. Finally, taking into account the I/O complexity $\mathcal{O}((n/B)\log_{M/B}(n/B))$ of the final multiway merging gives the following result.

Theorem 3.3. *Given the text of length n over an alphabet of size σ, the associated suffix array can be computed with the algorithm described above in*

$$\mathcal{O}\left(\frac{n^2}{M}\log\left(2 + \frac{\log\sigma}{\log\log n}\right)\right) \text{ time}$$

and

$$\mathcal{O}\left(\frac{n^2\log\sigma}{MB\log n} + \frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B}\right) \text{ I/Os.}$$

To simplify the I/O complexity, we make a (weak) assumption that $M \geq B\log_\sigma n$. Then, either $n \leq M^2/B$ and the I/O complexity is $\mathcal{O}(n/B)$ or $n > M^2/B$ in which case the first term always dominates. Thus, when $M \geq B\log_\sigma n$ the I/O complexity simplifies to

$$\mathcal{O}\left(\frac{n}{B}\left(1 + \frac{n\log\sigma}{M\log n}\right)\right).$$

The algorithm of Ferragina et al. (modified to compute SA rather than BWT) has the I/O complexity $\mathcal{O}\big(n^2/(MB)\big)$, which is dominated by partial suffix arrays merging. Our improved algorithm with delayed merging thus improves the I/O complexity by a factor $\log_\sigma n$.

3.2.6 Implementation details

For a concrete analysis, assume that $m \leq 2^{32}$, $n \leq 2^{40}$, and $\sigma \leq 256$. Thus, all positions in the text can be encoded using 40-bit integers.

To determine the exact value of constant M/m, we now have a look at RAM consumption during different steps of segment processing.

The first step is the construction of $\mathsf{SA}_{\mathsf{X:Y}}$, which from Lemma 3.2 is equivalent to suffix-sorting the modified string X'. For that task, we use divsufsort [60], a highly optimized suffix array construction algorithm by Yuta Mori. It essentially requires no extra space, thus we only need to keep X' and the output suffix array in RAM, for a total of $5m$ bytes.

The second step is the construction of the gap array. In this step we need to simultaneously keep the gap array $\mathsf{gap}_{\mathsf{X:Y}}$ and the rank data structure built over $\mathsf{BWT}_{\mathsf{X:Y}}$ in RAM. To compute $\mathsf{BWT}_{\mathsf{X:Y}}$ we scan $\mathsf{SA}_{\mathsf{X:Y}}$ and overwrite each entry $\mathsf{SA}_{\mathsf{X:Y}}[i]$ with $\mathsf{BWT}_{\mathsf{X:Y}}[i]$. The resulting $\mathsf{BWT}_{\mathsf{X:Y}}$ is then copied into the array storing the current text segment (which is no longer needed) and $\mathsf{SA}_{\mathsf{X:Y}}$ is released.

To represent the gap array we use the fact that although single gap values can be very large, the average value is not larger than n/m. Consequently, we only reserve m bytes for the gap array. Whenever the algorithm attempts to increment the gap entry $\mathsf{gap}_{\mathsf{X:Y}}[i] = 255$, we set $\mathsf{gap}_{\mathsf{X:Y}}[i] = 0$ and store i into a separate list of wrapped-around counters E. The size of E in the worst case is $n/64$ bytes, but in most practical situations it is very small. Only the most recent elements of E are stored in RAM in a small buffer. The buffer is flushed to disk, whenever it gets full and thus handling E requires a negligible amount of RAM. This gap representation is the second improvement of Paper I over the algorithm of Ferragina et al. [22]. In their algorithm each gap value is represented using 32 bits.

A space-efficient implementation of the gap array allows using the rank data structure of size $4m$ without increasing the peak RAM usage of $5m$ needed to store X and $\mathsf{SA}_{\mathsf{X:Y}}$ in the first step. In our implementation we use an improved version of rank data structure of Ferragina et al., which is the third contribution of Paper I. Our improvements are based on the alphabet partitioning technique [7] and fixed block boosting [45]. The resulting data structure is about 40% faster than the original and uses less than $4.2m$ bytes of RAM. Thus in total, the peak RAM usage when processing a segment of

size m is $5.2m$. As a result, our algorithm uses about 35% larger segments (compared the algorithm of Ferragina et al.), i.e., the number of segments (and thus overall runtime) is decreased by 35%.

The disk space usage peak occurs during the last stage, when we merge the partial suffix arrays into the final suffix array, and consists of

- n bytes for input text,
- $4n$ bytes for partial suffix arrays,
- $5n$ bytes for the output suffix array,
- $1.5n$ bytes for the vbyte-encoded [80] gap arrays (assuming $m \leq 128n$. If this is not true, the gap can take more space, but for such ratio n/m the algorithm is not competitive with other algorithms in terms of speed).

In total, the disk space usage is about $11.5n$ bytes. It can be reduced, if the partial suffix arrays are stored in small files, which are deleted immediately after they are processed. The free space is reclaimed by the output suffix array, i.e., the merging is performed almost in-place. This results in the reduction of the peak disk space usage to about $6.5n$ bytes. This improvement was not included in the version used for experiments in Paper I but has been subsequently incorporated into the parallel version of the SAscan, which we discuss at the end of this chapter.

3.3 Practical performance

In Paper I we performed the experimental comparison of SAscan to eSAIS, the best suffix-sorting algorithm in previous studies [10]. The SAscan algorithm was implemented as described in Section 3.2.6. In particular, we use 40-bit integers to represent the positions in the string, which is consistent with the version of eSAIS used in experiments.

The experiments were performed on a variety of different inputs and two different hardware configurations. Each of our improvements:

- replacing the 32-bit gap array with an 8-bit representation and thus reducing the number of segments by 35%,
- improving the rank data structure by applying the alphabet partitioning [7] and fixed block boosting [45] techniques,
- delaying the merging of partial suffix arrays,

results in a significant runtime reduction. The combined effect of our improvements was a speedup by a factor of about 3.5 compared to the BWT

construction algorithm of Ferragina et al. Compared to eSAIS, SAscan is faster up to a point, where the text is 4.7–7.4 times the size of internal memory.

3.4 Parallelizing the computation

The algorithm presented in this section is the fastest way to build the suffix array but only up to a point, where the text is about five times the RAM size. The runtime of the algorithm is, however, not I/O-bound, but dominated by the rank queries during gap array construction. This motivated us to investigate the SAscan algorithm in the multi-core setting and has led to a very substantial further speed-up. On a machine with 12 physical cores, the parallel implementation of SAscan, called pSAscan [43], moves the point, up to which the algorithm is faster than eSAIS to about 80 times the size of RAM. This allowed us to suffix-sort a 1 TiB file in a little over 8 days using 7.2 TiB of disk space.

On the way to this result, we also implemented the in-RAM parallel version of SAscan which turned out to be faster (by a factor of about two) and more space efficient (by a factor of at least two) than the previously best internal-memory parallel suffix-sorting algorithm.

Some of the experiments in this thesis (see Chapter 6) were revised to include, in addition to SAscan, also its parallel version pSAscan.

Chapter 4

LCP array construction

In many applications, the suffix array needs to be augmented with the longest-common-prefix (LCP) array, which stores the lengths of longest common prefixes between lexicographically adjacent suffixes.

The LCP array construction in internal memory is a very well studied topic. The historically first algorithm runs in linear time but uses a lot of space ($3n$ integers in addition to the input text). Many algorithms followed, aiming at either improving the space or runtime [56, 59, 71, 44, 74, 35, 9, 26]. Some of the algorithms (e.g., [71, 44]) can be made semi-external, i.e., they only require that the text is kept in the main memory and all remaining data structures are efficiently stored and accessed from the secondary memory. The fully external-memory construction, however, remains a problem.

The only previously known way to compute the LCP array in external memory is to use an external memory suffix array construction algorithm modified to compute the LCP array as a by-product [47, 10]. While this modification does not affect the time or I/O complexity of the algorithm, it significantly increases the running time and, more importantly, disk space usage. So much, in fact, that the disk space usage becomes the main limitation in the scalability of these algorithms. For example, the disk space usage of eSAIS [10] – currently the fastest external-memory algorithm computing both suffix and LCP array – is $54n$ bytes for a text of n bytes.

In this chapter we describe the first standalone algorithm constructing the LCP array in external memory. The algorithm, called LCPscan, does not augment the suffix array construction, but takes the text and the suffix array as an input, thus it can be combined with any suffix array construction algorithm, including a better one developed in the future. LCPscan can be seen as an external-memory version of the internal-memory algorithm of [44]. In our experiments, LCPscan is faster and uses less than a quarter of the disk space, compared to eSAIS. The material is based on Paper II.

i	0	1	2	3	4	5	6	7	8	9	10	11
$\mathsf{T}[i]$	b	a	b	a	a	b	b	a	b	b	a	b
$\mathsf{SA}[i]$	3	10	1	7	4	11	2	9	0	6	8	5
$\mathsf{ISA}[i]$	8	2	6	0	4	11	9	3	10	7	1	5
$\Phi[i]$	9	10	11	12	7	8	0	1	6	2	3	4
$\mathsf{LCP}[i]$	0	1	2	2	5	0	1	2	3	3	1	4
$\mathsf{PLCP}[i]$	3	2	1	0	5	4	3	2	1	2	1	0
$i + \mathsf{PLCP}[i]$	3	3	3	3	9	9	9	9	9	11	11	11

Table 4.1: Examples of the arrays used by the algorithm for the text $\mathsf{T} =$ babaabbabbab.

4.1 Preliminaries

The key to almost all efficient algorithms constructing the LCP array is the computation of LCP values in text order rather than in the lexicographical order. The *permuted LCP array* $\mathsf{PLCP}[0..n)$ is the LCP array permuted from the lexicographical order into the text order. Formally, for $i \in [0..n)$,

$$\mathsf{PLCP}[\mathsf{SA}[i]] = \mathsf{LCP}[i].$$

It is useful to also consider the alternative definition: $\mathsf{PLCP}[j] = \mathsf{lcp}(j, \Phi[j])$, $j \in [0..n)$, where $\Phi[0..n)$ is defined as $\Phi[\mathsf{SA}[i]] = \mathsf{SA}[i-1]$, $i \in [1..n)$ and $\Phi[\mathsf{SA}[0]] = n$. The key property of the PLCP array is summarized in the following Lemma (see Table 4.1 for an example).

Lemma 4.1 ([48, 44])**.** *For $j \in [1..n)$, $\mathsf{PLCP}[j] \geq \mathsf{PLCP}[j-1] - 1$.*

4.2 The Φ algorithm

Our starting point is the internal-memory algorithm described by Kärkkäinen et al. [44]. It is currently the fastest algorithm to compute the LCP array in RAM, given the text and suffix array. The algorithm is given in Figure 4.1

The algorithm starts by computing the Φ array from SA. Next, it computes all PLCP values using Lemma 4.1. This ensures that the total number of steps in the while loop in line 5 is $\mathcal{O}(n)$ and thus the whole algorithm runs in linear time. Finally, it permutes PLCP to LCP array.

4.3 The new algorithm

On a high level, our new algorithm is an external memory version of the Φ algorithm. Let us now have a look in detail at how each of the steps of the Φ algorithm can be adapted to external memory.

Algorithm $\Phi(\mathsf{T}, \mathsf{SA})$
// Step I
1: $\Phi[\mathsf{SA}[0]] \leftarrow n$
2: **for** $i \leftarrow 1$ **to** $n-1$ **do**
3: $\Phi[\mathsf{SA}[i]] \leftarrow \mathsf{SA}[i-1]$
// Step II
4: $\ell \leftarrow 0$
5: **for** $j \leftarrow 0$ **to** $n-1$ **do**
6: **while** $\max(j, \Phi[j]) + \ell < n$ and $\mathsf{T}[j+\ell] = \mathsf{T}[\Phi[j]+\ell]$ **do**
7: $\ell \leftarrow \ell + 1$
8: $\mathsf{PLCP}[j] \leftarrow \ell$
9: $\ell \leftarrow \max(\ell - 1, 0)$
// Step III
10: **for** $i \leftarrow 0$ **to** $n-1$ **do**
11: $\mathsf{LCP}[i] \leftarrow \mathsf{PLCP}[\mathsf{SA}[i]]$

Figure 4.1: The Φ algorithm.

First, observe that Step I and III are easy to implement in external memory using sorting. In Step I we scan the suffix array creating a triple $(\mathsf{SA}[i], \mathsf{SA}[i-1], i)$ for every $i \in [1..n)$. The triples are sorted by the first component to obtain sequence $(j, \Phi[j], \mathsf{ISA}[j])$. For the third step we observe that the instruction $\mathsf{LCP}[i] \leftarrow \mathsf{PLCP}[\mathsf{SA}[i]]$ is equivalent to $\mathsf{LCP}[\mathsf{ISA}[j]] \leftarrow \mathsf{PLCP}[j]$, thus we can implement this step by sorting the pairs $(\mathsf{ISA}[j], \mathsf{PLCP}[j])$ by the first component. The resulting sequence is $(i, \mathsf{LCP}[i])$.

Let us have a closer look at Step II. The access to PLCP and Φ array, and the first access to T in line 6 are sequential and thus can be efficiently implemented in external memory. The main issue is the second access to the text in line 6, since the value $\Phi[j] + \ell$ changes when j changes.

Our solution is to divide the text into $\lceil n/m \rceil$ segments, each of size m (except possibly the last one), where m is such that m characters of text fit in RAM. Each of the segments is loaded once into the main memory. We organize the computation, so that when the segment $\mathsf{T}[s..e)$ is loaded into RAM, the random accesses to the text are restricted only to interval $[s..e)$.

4.3.1 Eliminating random access to text

When processing the segment $\mathsf{T}[s..e)$, we first sort all pairs $(j, \Phi[j])$, $j \in [s..e)$ by the second component. In the rest of the algorithm, the pairs

Procedure `ProcessSegment`$(s, e, \mathsf{T}, \Phi[s..e))$
1: $\mathsf{Q} \leftarrow \{(j, \Phi[j]) : j \in [s..e)\}$
2: sort Q by the second component
3: $\Phi_{\text{prev}} \leftarrow 0$, $\ell_{\text{prev}} \leftarrow 0$
4: **for** $(j, \Phi[j]) \in \mathsf{Q}$ **do**
5: $\quad \ell \leftarrow \max(0, \Phi_{\text{prev}} + \ell_{\text{prev}} - \Phi[j])$
6: $\quad$ **while** $\max(j, \Phi[j]) + \ell < n$ **and** $\mathsf{T}[j + \ell] = \mathsf{T}[\Phi[j] + \ell]$ **do**
7: $\quad\quad \ell \leftarrow \ell + 1$
8: $\quad \mathsf{L} \leftarrow \mathsf{L} \cup \{(j, \ell)\}$
9: $\quad \Phi_{\text{prev}} \leftarrow \Phi[j]$, $\ell_{\text{prev}} \leftarrow \ell$
10: sort L by the first component
11: $\mathsf{PLCP}[s..e) \leftarrow \{\ell : (j, \ell) \in \mathsf{L}\}$
12: **return** $\mathsf{PLCP}[s..e)$

Figure 4.2: Processing of the text segment $\mathsf{T}[s..e)$ in LCPscan.

$(j, \Phi[j])$ are processed in this order.

Suppose we have just completed the lcp computation for the pair $(j, \Phi[j])$. Let $\ell = \mathsf{lcp}(j, \Phi[j])$, i.e., the last symbol comparison was between $\mathsf{T}[j + \ell]$ and $\mathsf{T}[\Phi[j] + \ell]$. Let $(j', \Phi[j'])$ be the next pair to be processed. We have $\Phi[j'] > \Phi[j]$, but not necessarily $\Phi[j] + \ell \leq \Phi[j']$, thus to access $\mathsf{T}[\Phi[j']]$ we might need to go back in the text. This can be avoided using the following generalization of Lemma 4.1.

Lemma 4.2. *Let $j, j' \in [0..n)$. If $j \leq j'$, then $j + \mathsf{PLCP}[j] \leq j' + \mathsf{PLCP}[j']$. Symmetrically, if $\Phi[j] \leq \Phi[j']$, then $\Phi[j] + \mathsf{PLCP}[j] \leq \Phi[j'] + \mathsf{PLCP}[j']$.*

Proof. The claim of Lemma 4.1 is equivalent to $j + \mathsf{PLCP}[j] \leq (j + 1) + \mathsf{PLCP}[j + 1]$. Iteratively applying this result gives the first claim. The second claim follows by symmetry. □

The above Lemma gives $\Phi[j] + \ell \leq \Phi[j'] + \mathsf{lcp}(j', \Phi[j'])$, thus when computing $\mathsf{lcp}(j', \Phi[j'])$ we can skip the first $\max(0, \Phi[j] + \ell - \Phi[j'])$ symbols and continue where we left off when processing the pair $(j, \Phi[j])$.

Therefore, processing the pairs in this order restricts random access to text only to segment $\mathsf{T}[s..e)$, but at the cost of scanning the whole text to implement the remaining sequential access. The pseudo-code of the procedure processing segment $\mathsf{T}[s..e)$ is given in Figure 4.2.

Note that after changing the order in which we process pairs $(j, \Phi[j])$, we no longer obtain the PLCP values in correct order, thus first we collect them in a set L (line 8) and sort at the end of the procedure (lines 10–11).

4.3.2 Handling long LCPs

The above algorithm guarantees that for any pair $(j, \Phi[j])$ processed in lines 4–9 we have $j \in [s..e)$. However, when $\mathsf{lcp}(j, \Phi[j])$ is larger than $e - j$ we need to access symbols past the text segment $\mathsf{T}[s..e)$. To prevent this, we modify the algorithm as follows.

Whenever during processing a pair $(j, \Phi[j])$ we have $j + \ell = e$ in line 6, we pause the lcp computation for that pair and store $(j, \Phi[j])$ in the set R_e. The computation of the pairs stored in R_e is resumed when we start processing the segment beginning at position e. Formally, at the end of processing the segment $\mathsf{T}[s..e)$ we have

$$\mathsf{R}_e = \{(j, \Phi[j]) : j < e \text{ and } j + \mathsf{lcp}(j, \Phi[j]) \geq e\}.$$

Therefore, the procedure for processing the segment $\mathsf{T}[s..e)$ now also receives R_s as an extra input from the previous segment. It is already sorted by the second component, hence it can be easily merged on-the-fly with Q. More precisely, in line 4, we either take the first element of Q or R_s, depending on which one has a smaller second component, and continue the lcp computation. If the next processed element is $(j, \Phi[j]) \in \mathsf{R}_s$, we set $\ell' \leftarrow s - j$ when we start processing the triple. Otherwise, we set $\ell' \leftarrow 0$. Line 5 becomes:

$$5 : \ell \leftarrow \max(\ell', \Phi_{\text{prev}} + \ell_{\text{prev}} - \Phi[j]).$$

This way, the computation of $\mathsf{lcp}(j, \Phi[j])$ is resumed for elements of R_s, rather than started from scratch. The computation of pairs from Q is not affected.

The output of processing $\mathsf{T}[s..e)$ is no longer $\mathsf{PLCP}[s..e)$, because some lcp computations were postponed to the next segment. However, Lemma 4.2 implies that if $(j, \cdot) \in \mathsf{R}_e$ then $(j', \cdot) \in \mathsf{R}_e$ for any $j' > j$, thus the output is in fact $\mathsf{PLCP}[s'..e')$ for some $s' \leq s$ and $e' \leq e$, i.e., we still obtain values of PLCP in correct order.

4.3.3 Reducing the number of boundary crossings

The size of R_s and R_e is $\Theta(n)$ in the worst case, since a single lcp value can span several segments. Thus, over the whole algorithm, the total size of these sets can be as large as $\Theta(n^2/m)$, e.g., for $\mathsf{T} = \mathsf{a}^n$. This translates to enormous extra I/O, since these sets contain pairs of integers.

To prevent this, we will use the technique of irreducible lcp values introduced in [44].

Definition 4.3. *An lcp value* $\mathsf{PLCP}[j]$ *is said to be reducible if* $\mathsf{T}[j-1] = \mathsf{T}[\Phi[j]-1]$. *Otherwise, in particular when* $j = 0$ *or* $\Phi[j] \in \{0, n\}$, *the value is irreducible.*

The following Lemma shows that it is easy to compute all reducible lcp values once irreducible lcp values are known.

Lemma 4.4 ([44])**.** *If* $\mathsf{PLCP}[j]$ *is reducible then* $\mathsf{PLCP}[j] = \mathsf{PLCP}[j-1]-1$.

Furthermore, Kärkkäinen et al. [44] have shown that the sum of irreducible lcp values is at most $2n \log n$. The constant factor was later improved by Kärkkäinen et al. [40].

Lemma 4.5 ([40])**.** *The sum of all irreducible lcp values is* $\leq n \log n$.

The above tools are used to modify the algorithm as follows. When creating the set Q in line 1 of the algorithm in Figure 4.2, we only include the pairs $(j, \Phi[j])$ such that $\mathsf{PLCP}[j]$ is irreducible. The remaining (reducible) values can be easily computed in line 11 using Lemma 4.4.

To determine all j such that $\mathsf{PLCP}[j]$ is irreducible we use the following Lemma (we omit the proof that can be found in Paper II).

Lemma 4.6. $\mathsf{PLCP}[j]$ *is reducible iff* $j > 0$ *and* $\Phi[j-1] = \Phi[j] - 1$ *and* $\mathsf{PLCP}[j-1] > 0$.

The first and second condition can be checked while scanning $\Phi[j]$ in line 1 of Figure 4.2. All positions satisfying the third condition can be precomputed, since $\mathsf{PLCP}[j-1] = 0$ implies that $j-1$ is the starting position of the lexicographically smallest suffix starting with $\mathsf{T}[j-1]$. There is at most σ such suffixes and they can be computed from SA if the symbol frequencies in T are known. To compute the symbol frequencies we scan (for small σ) or sort (for large σ) the text.

With the above modification, the number of segment boundary crossings is reduced to $\mathcal{O}(n + (n \log n)/m)$ (see Lemma 4.5), which is $\mathcal{O}(n)$ under a reasonable assumption $m = \Omega(\log n)$. Therefore, the total size of sets R_s and R_e over the whole algorithm is $\mathcal{O}(n)$.

4.3.4 Theoretical analysis

We now analyse the complexity of the algorithm in the standard external memory model.

Theorem 4.7. *Given the text of length n over an alphabet of size σ and its suffix array, the associated LCP array can be computed with the algorithm described above in*

$$\mathcal{O}\left(\frac{n^2}{M(\log_\sigma n)^2} + n \log_{\frac{M}{B}} \frac{n}{B}\right) \text{ time}$$

and

$$\mathcal{O}\left(\frac{n^2}{MB(\log_\sigma n)^2} + \frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B}\right) \text{ I/Os}$$

in the standard external memory model.

Proof. The text is divided into segments of size m, where m is chosen so that m text characters can fit in RAM. The size of RAM is $M \log n$ bits, thus we can fit $m = \Theta(M \log_\sigma n)$ symbols in that space. For each segment, we scan the whole text, which takes $\mathcal{O}(n/(B \log_\sigma n))$ I/Os. To estimate the number of operations, we observe that comparing ℓ text symbol takes $\mathcal{O}(1 + \ell/\log_\sigma n)$ time, assuming we pack $\log_\sigma n$ symbols into a machine word. For a single segment, this amounts to $\mathcal{O}(m + n/\log_\sigma n)$ operations in total. This gives the first term in the complexities.

All other operations in the algorithm involve streaming and sorting tuples of integers. The total number of elements involved in each of these operations is $\mathcal{O}(n)$. This gives the second term in the complexities. □

4.3.5 Implementation details

In the implementation of the algorithm used in experiments we assume that each text symbol is represented using 1 byte and each integer needs 5 bytes. The implementation is therefore capable of handling texts up to 1 TiB.

For external memory sorting we use the STXXL library [19]. The library operates using the concept of pipelining. That is, rather than writing the result of one step of computation to disk and loading it into RAM at the beginning of the next step, we perform two steps simultaneously: the output of the first step is directly fed as an input to the next stage.

All steps of the computation in LCPscan are implemented using the above methodology. For example, when we obtain Φ values as the second component of triples $(j, \Phi[j], \mathsf{ISA}[j])$ in the first step, we do not write $\Phi[j]$ values to disk. Instead, we immediately form the pairs $(j, \Phi[j])$ which are then sorted by the second component.

Using pipelining all through the algorithm, and assuming that the external memory sorting requires only a single pass over the input, the I/O volume of LCPscan is $71n + 40r + \lceil n/m \rceil n$ bytes, where r is the number of irreducible lcp values.

The peak disk space usage occurs during the step described above, when we read the triples $(j, \Phi[j], \mathsf{ISA}[j])$ from disk and write $\mathsf{ISA}[j]$ and pairs $(j, \Phi[j])$ sorted by the second component to disk. All that data takes $30n$ bytes of disk space in the worst case. In addition we have the suffix array and text on disk occupying $6n$ bytes, for a total of $36n$ bytes.

Reducing disk space usage. To reduce the peak disk space usage, we split the text into q parts of sizes $n_1, n_2, \ldots, n_q$ and execute the algorithm separately for each part. More precisely, in the first step we scan SA and form a triple $(\mathsf{SA}[i], \mathsf{SA}[i-1], i)$ only if $\mathsf{SA}[i]$ belongs to the current part. The rest of the algorithm can be easily modified, because the main data structures used in the algorithm (ISA, Φ, PLCP) are all indexed in text order, i.e., we simply compute and store contiguous ranges of these arrays rather than full arrays.

The output of the algorithm for a single part is a subsequence of the LCP array containing only the entries $\mathsf{LCP}[i]$ such that $\mathsf{SA}[i]$ belongs to the current part. Once all parts have been processed, they are merged using SA to determine the order.

The disadvantage of this partial processing is that enumerating the triples that belong to the currently processed part, requires scanning the whole suffix array. These extra scans and the merging of LCP subsequences add $15n + 5(q-1)$ bytes to the I/O volume, e.g., for $q = 4$, the I/O volume is $101n + 40r + \lceil n/m \rceil n$.

With the above technique, the peak disk space usage during the most disk-demanding step is reduced to $6n + 5\sum_{i=1}^{j-1} n_i + 30n_j$ for the j^{th} part. The optimal division of text into q parts (i.e., the division that minimizes the maximum of the above expression over all parts) is:

$$\begin{aligned} n_1 &= \frac{6^{q-1}}{6^q - 5^q} n, \\ n_2 &= \frac{5}{6} n_1, \\ &\ldots \\ n_q &= \frac{5}{6} n_{q-1}. \end{aligned}$$

With $q = 10$ the peak disk space usage of LCPscan is reduced to less than $12n$. This is only n bytes more than is required for input (suffix array and text occupying $6n$ bytes in total) and output (the LCP array occupying $5n$ bytes).

Algorithm	Runtime	Peak disk usage	I/O volume
eSAIS (SA only)	5.0 days	2.8 TiB	30.5 TiB
pSAscan	2.2 days	0.9 TiB	16.1 TiB
LCPscan	1.6 days	1.9 TiB	16.9 TiB
eSAIS (SA+LCP)	9.9 days	6.0 TiB	60.3 TiB
eSAIS (SA) + LCPscan	6.6 days	3.4 TiB	47.5 TiB
pSAscan + LCPscan	3.8 days	1.9 TiB	33.0 TiB

Table 4.2: Summary of experiments on the 120 GiB testfile (Wikipedia XML) using 3.5 GiB of RAM.

4.4 Practical performance

We implemented the LCPscan algorithm with all the optimizations described above. Since there are no other existing standalone methods for LCP array construction in external memory, we compare it to eSAIS [10], the fastest algorithm computing both the suffix and LCP array in external memory. eSAIS can also compute only the suffix array. In our experiments we measure the difference in runtime between these two modes and compare to LCPscan. In all experiments with LCPscan we used $q = 10$, i.e., the input was processed in ten parts. Paper II contains detailed experimental results. For all files, LCPscan is faster than eSAIS by a factor 2–4, while simultaneously using less than a quarter of disk space.

In the second experiment we assume that we start the computation with the text only. The results are given in Table 4.2. The combination of LCPscan and eSAIS (used only for construction of SA) achieves a 33% speed-up and 43% disk space usage reduction, compared to eSAIS computing both suffix and LCP array.

Separating the LCP array construction from suffix sorting allows replacing eSAIS with other algorithm. Using the recent parallel suffix array construction algorithm called pSAscan (see Section 3.4), we obtained further improvement in runtime and disk space usage by about 44%.

Additional experiments showing the advantage of separating the LCP array computation from suffix sorting (in the context of external memory Lempel-Ziv parsing) are shown in Chapter 6.

Chapter 5

LZ77 parsing in internal memory

Lempel-Ziv (LZ77) parsing is a method of encoding the text that takes advantage of fragments occurring in several positions to achieve compression. The method is used in several popular lossless compressors such as gzip or 7-zip [70]. The parsing also has applications in other domains, such as efficient computation of all repetitions in a string [51] or approximation of the smallest context-free grammar [13].

In recent years LZ77 has received increasing attention due to a new problem, namely, indexing highly repetitive collections. Such data is abundant nowadays: genomic collections produced by high-throughput sequencing machines [20, 57], versioned collections of source code and multi-author documents, such as Wikipedia [75] or web crawls [24].

LZ77 exploits a high degree of repetitions particularly well, and several LZ77-based or grammar-based (where grammar can be derived from LZ77 parsing) indexes have been proposed [54, 30, 31, 21, 32]. In many of the above applications, including index construction, computing the parsing is a bottleneck [54, 32].

Time-optimal solutions to LZ77 parsing have been known for years [3, 18, 16] (see also the recent survey [5] for the overview of the prior art), but the increasing need for efficient parsing algorithms has recently spawned several new algorithms. Many of the solutions [69, 81, 76, 52, 8] focus on achieving a good time complexity, $\mathcal{O}(n \cdot \mathrm{polylog}(n))$, while using a space close to what is necessary to hold the text, i.e., $\mathcal{O}(n \log \sigma)$ bits. These algorithms usually rely on sophisticated data structures and thus are mostly of theoretical interest.

A considerable amount of effort also went into developing algorithms that are primarily concerned with practical performance [14, 67, 38, 37]. In this thesis we focus on that group. This chapter gives a unified view

of existing and several new practical algorithms for computing the LZ77 parsing that improve upon the prior solutions in terms of time and space. For now, we restrict the discussion to internal-memory algorithms. Using disk space in the LZ77 parsing is the topic of the next chapter. The material in this chapter is based on Papers III and IV.

5.1 Preliminaries

LZ77 parsing via LPF array. One way to compute the LZ77 parsing is to compute all LPF values $\mathsf{LPF}[0..n)$ and with a single scan select the ones in the parsing. Crochemore and Ilie describe two algorithms using this approach [16, 18]. Both algorithms take the suffix array as an input. The main difference is that the first algorithm uses direct symbol comparisons to compute lcp values and thus requires access to the text. The second algorithm takes, in addition to SA, the LCP array as an input and thus does not require access to the text at all. Both algorithms run in $\mathcal{O}(n)$ time. Ohlebusch and Gog [67] have shown how the second algorithm can be improved by interleaving the computation of the LCP and the LPF array. The modification does affect the time complexity, but the resulting algorithm runs faster in practice.

Lazy LZ77 parsing. It is known that the number of phrases in the LZ77 parsing, denoted by z, satisfies $z = \mathcal{O}(n/\log_\sigma n)$ [39]. Therefore, unless the alphabet is very large, we have $z = o(n)$ and most of the LPF pairs are not used in the parsing. Skipping the computation of these unused LPF pairs could speed up the LZ77 parsing, since typically computing each LPF pair attracts at least one cache miss. However, the exact positions that can be omitted cannot be identified ahead of time. They can only be detected, if we postpone the computation of lcp values until it is absolutely necessary. This approach is known as *lazy Lempel-Ziv parsing* [49, 37, 41].

The main tools used by the algorithms using the lazy approach are *next and previous smaller value (NSV/PSV)* arrays. For $i \in [0..n)$, let

$$\mathsf{NSV}_{\mathsf{lex}}[i] = \min\{i' \in [i+1..n) \mid \mathsf{SA}[i'] < \mathsf{SA}[i]\},$$
$$\mathsf{PSV}_{\mathsf{lex}}[i] = \max\{i' \in [0..i) \mid \mathsf{SA}[i'] < \mathsf{SA}[i]\}.$$

If any of the sets on the right is empty we set $\mathsf{NSV}_{\mathsf{lex}}[i] = -1$ and $\mathsf{PSV}_{\mathsf{lex}}[i] = -1$. The above arrays are defined with respect to the suffix array, i.e., in

Algorithm `Parse`
1: $j \leftarrow 0$
2: **while** $j < n$ **do**
3: $nsv \leftarrow$ compute $\mathsf{NSV}_{\mathsf{text}}[j]$
4: $psv \leftarrow$ compute $\mathsf{PSV}_{\mathsf{text}}[j]$
5: $j \leftarrow \texttt{Factor}(j, nsv, psv)$

Procedure `Factor`(j, nsv, psv)
1: $\ell_{nsv} \leftarrow \mathsf{lcp}(j, nsv)$
2: $\ell_{psv} \leftarrow \mathsf{lcp}(j, psv)$
3: **if** $\ell_{nsv} > \ell_{psv}$ **then**
4: $(p, \ell) \leftarrow (nsv,\ \ell_{nsv})$
5: **else**
6: $(p, \ell) \leftarrow (psv,\ \ell_{psv})$
7: **if** $\ell = 0$ **then** $p \leftarrow \mathsf{T}[j]$
8: **output** factor (p, ℓ)
9: **return** $j + \max(1, \ell)$

Figure 5.1: Computing the LZ77 parsing from $\mathsf{NSV}_{\mathsf{text}}$ and $\mathsf{PSV}_{\mathsf{text}}$ values. We assume that $\mathsf{lcp}(j, j')$ returns 0 if $j' \notin [0..n)$.

lexicographical order. It is useful to also define them in text order:

$$\mathsf{NSV}_{\mathsf{text}}[j] = \mathsf{SA}[\mathsf{NSV}_{\mathsf{lex}}[\mathsf{ISA}[j]]], \tag{5.1}$$

$$\mathsf{PSV}_{\mathsf{text}}[j] = \mathsf{SA}[\mathsf{PSV}_{\mathsf{lex}}[\mathsf{ISA}[j]]]. \tag{5.2}$$

If $\mathsf{NSV}_{\mathsf{lex}}[\mathsf{ISA}[j]] = -1$ (or $\mathsf{PSV}_{\mathsf{lex}}[\mathsf{ISA}[j]] = -1$), we set $\mathsf{NSV}_{\mathsf{text}}[j] = -1$ ($\mathsf{PSV}_{\mathsf{text}}[j] = -1$). The NSV/PSV arrays can be used to compute the LPF pairs of the text using the following Lemma. Recall, that we denote $\mathsf{LPF}[j] = (p_j, \ell_j)$.

Lemma 5.1 ([18]). *For any $j \in [0..n)$, it holds $\ell_j = \max\{\ell_{\text{nsv}}, \ell_{\text{psv}}\}$, where $\ell_{\text{nsv}} = \mathsf{lcp}(j, \mathsf{NSV}_{\mathsf{text}}[j])$ and $\ell_{\text{psv}} = \mathsf{lcp}(j, \mathsf{PSV}_{\mathsf{text}}[j])$.*

The algorithm for computing the LZ77 parsing from $\mathsf{NSV}_{\mathsf{text}}$ and $\mathsf{PSV}_{\mathsf{text}}$ arrays is given in Figure 5.1. Whenever the procedure `Factor` returns after performing ℓ symbols comparisons, j is advanced by at least $\ell/2$ in line 5 of `Parse`, thus the algorithm runs in $\mathcal{O}(n)$ time.

The problem of computing the LZ77 parsing can therefore be reduced to computing $\mathsf{NSV}_{\mathsf{text}}/\mathsf{PSV}_{\mathsf{text}}$ values.

5.2 Precomputing NSV/PSV

Our first approach to implement access to NSV/PSV arrays is to simply precompute all values. This idea was first proposed by Goto and Bannai [37]. They describe three new linear-time algorithms, which all first compute either $\mathsf{NSV}_{\mathsf{text}}/\mathsf{PSV}_{\mathsf{text}}$ or $\mathsf{NSV}_{\mathsf{lex}}/\mathsf{PSV}_{\mathsf{lex}}$ arrays and then use them to compute the parsing. The fastest of the three algorithms is called BGS. The first contribution of Paper III is the improved version of the BGS algorithm. Our changes do not affect the time complexity, but make the algorithm faster and more space efficient.

Algorithm `ComputeNSV/PSV`

```
1:  SA[−1] ← −1   // bottom of the stack
2:  SA[n] ← −1    // empties the stack at the end
3:  top ← −1      // top of the stack
4:  for i ← 0 to n do
5:      while SA[top] > SA[i] do
6:          NSV_text[SA[top]] ← SA[i]
7:          PSV_text[SA[top]] ← SA[top − 1]
8:          top ← top − 1   // pop from stack
9:      top ← top + 1
10:     SA[top] ← SA[i]   // push to stack
```

Figure 5.2: Computation of $\mathsf{NSV}_{\mathsf{text}}$ and $\mathsf{PSV}_{\mathsf{text}}$ arrays.

The first step of the algorithm is the computation of SA for the input text. This can be done in linear time and $(1+\epsilon)n \log n$ bits of working space[1] (for any $\epsilon > 0$), including the space for output SA and assuming integer alphabet [47].

The next step is the computation of the $\mathsf{NSV}_{\mathsf{text}}/\mathsf{PSV}_{\mathsf{text}}$ values from the text and SA. The algorithm is given in Figure 5.2. To simplify the pseudocode, we assume that $\mathsf{SA}[-1]$ and $\mathsf{SA}[n]$ are valid addresses. The algorithm scans the suffix array while maintaining the stack of suffixes. After processing suffix $\mathsf{SA}[i]$, the content of the stack (top to bottom) is: $\mathsf{SA}[i]$, $\mathsf{PSV}_{\mathsf{text}}[\mathsf{SA}[i]]$, $\mathsf{PSV}_{\mathsf{text}}[\mathsf{PSV}_{\mathsf{text}}[\mathsf{SA}[i]]], \ldots, -1$. Our procedure differs from the one of Goto and Bannai in the following ways:

- We compute $\mathsf{NSV}_{\mathsf{text}}$ and $\mathsf{PSV}_{\mathsf{text}}$ directly, rather than computing $\mathsf{NSV}_{\mathsf{lex}}$ and $\mathsf{PSV}_{\mathsf{lex}}$ first and then permuting it with the help of ISA and SA.

- In the BGS algorithm the PSV value is stored whenever it is pushed to the stack. We delay this and always write NSV and PSV values together. This reduces the number of cache misses, since the arrays $\mathsf{NSV}_{\mathsf{text}}$ and $\mathsf{PSV}_{\mathsf{text}}$ are stored interleaved.

- Rather than storing the content of the stack in a separate array, we overwrite the suffix array with the stack. This is possible, because the stack is never larger than the already used part of SA.

[1] By working space we mean the space used by the algorithm in addition to the input and output. In some cases the working space includes the output and in those cases we explicitly mention that.

Algorithm `Parse`
1: **for** $i \leftarrow 0$ **to** $n-1$ **do**
2: $\quad \mathsf{ISA}[\mathsf{SA}[i]] \leftarrow i$
3: $j \leftarrow 0$
4: **while** $j < n$ **do**
5: $\quad i \leftarrow \mathsf{ISA}[j]$
6: $\quad nsv \leftarrow$ compute $\mathsf{NSV}_{\mathsf{lex}}[i]$
7: $\quad psv \leftarrow$ compute $\mathsf{PSV}_{\mathsf{lex}}[i]$
8: $\quad j \leftarrow \mathtt{Factor}(j, \mathsf{SA}[nsv], \mathsf{SA}[psv])$

Figure 5.3: Computing the LZ77 parsing with $\mathsf{NSV}_{\mathsf{lex}}$ and $\mathsf{PSV}_{\mathsf{lex}}$ values.

The new algorithm uses three integer arrays and thus requires $3n \log n$ bits of working space, which is $n \log n$ bits less than the BGS algorithm [2].

Further optimizations. Paper III describes several further improvements to the algorithm. For example, it is sufficient to only precompute the $\mathsf{PSV}_{\mathsf{text}}$ array in the preprocessing stage and then in the parsing stage the value $\mathsf{NSV}_{\mathsf{text}}$ can be obtained from SA and $\mathsf{PSV}_{\mathsf{text}}$, assuming we can overwrite $\mathsf{PSV}_{\mathsf{text}}$. The resulting algorithm uses only $2n \log n$ bits of working space.

The suffix array can be left untouched after the computation. This may be desirable if we want to reuse it later for other purpose. To achieve this, the top of the stack is stored in a fixed-size buffer (to save cache misses) while the rest is implicitly stored in the $\mathsf{PSV}_{\mathsf{text}}$ array.

5.3 NSV/PSV queries

In this section we describe a family of algorithms that abandon precomputed NSV/PSV arrays in favor of data structures capable of answering NSV/PSV queries on demand. This approach leads to one of the most space-efficient internal memory algorithms. The new family of algorithms is the second contribution of Paper III.

We start by rewriting the basic parsing algorithm in Figure 5.1 to use $\mathsf{NSV}_{\mathsf{lex}}/\mathsf{PSV}_{\mathsf{lex}}$ values instead of $\mathsf{NSV}_{\mathsf{text}}/\mathsf{PSV}_{\mathsf{text}}$ (see Eqs 5.1 and 5.2). The new version is given in Figure 5.3.

The PSV/NSV values in lines 6–7 can be found simply by scanning SA from $\mathsf{ISA}[j]$ in both directions until the value smaller than j is detected.

[2] We point out that one of the algorithms described by Goto and Bannai [37] also uses $3n \log n$ bits of working space but is notably slower than BGS for non-artificial inputs.

Despite the $\mathcal{O}(nz)$ worst-case time complexity, this approach works well on many types of data. However, an artificially designed input can cause the algorithm to slow down dramatically [49].

To prevent long scans of SA, we use the NSV/PSV data structure. Among the existing data structures for this task (see [25] for a theoretical overview), we found the practical data structure of Abeliuk et al. [2] to work best in our case. It uses $(n \log n)/\widehat{b}$ bits of space and answers queries in $\mathcal{O}(\widehat{b} + n/\widehat{b})$ time. For $\widehat{b} = \log n$ the algorithm in Figure 5.3 runs in $\mathcal{O}(n + z \log n)$ time and uses $2n \log n + n$ bits of working space. To simplify the analysis, in the remaining part of this section we assume $\widehat{b} = \log n$.

Reducing space usage. Computing and storing the full ISA requires significant time and space. In the parsing stage, however, only z elements of ISA are used. This can be exploited as follows. For some fixed constant $k > 0$ we store $\mathsf{ISA}[j]$ only if $j \bmod k = 0$. Those are called *sample* positions. Then, an arbitrary value of ISA can be computed using the so-called LF-mapping $\mathsf{LF}[0..n)$, which is defined with the equality

$$\mathsf{ISA}[j-1] = \mathsf{LF}[\mathsf{ISA}[j]].$$

Thus, for any $j \in [0..n)$, $\mathsf{ISA}[j] = \mathsf{LF}^s[\mathsf{ISA}[kj']]$, where $j' = \lceil j/k \rceil$ and $s = kj' - j \in [0..k)$. The problem therefore is reduced to the ability of answering LF queries fast and in small space.

Compact representations of LF-mapping are well studied in the context of compressed text indexes [63]. In most of these applications, typically only the Burrows-Wheeler transform $\mathsf{BWT}[0..n)$ is available (but not the plain suffix array). Then, the LF-mapping can be computed using the formula

$$\mathsf{LF}[i] = \mathsf{C}[\mathsf{BWT}[i]] + \mathsf{rank}_{\mathsf{BWT}[i]}(\mathsf{BWT}, i),$$

where $\mathsf{C}[c]$ is the total number of symbols smaller than c in $\mathsf{BWT}[0..n)$ and $\mathsf{rank}_c(\mathsf{BWT}, i)$ is the number of occurrences of c in $\mathsf{BWT}[0..i)$ (see [23]). In these applications, BWT is often replaced with wavelet trees, which support efficient access/rank queries (and much more, see the recent survey [62]).

The above approach does not, however, take any advantage of the suffix array in the uncompressed form, which is available in our algorithm. This additional information allows designing a faster and smaller representation of LF-mapping. Let $b > 0$ be a fixed constant and let $\mathsf{BWT}'[0..n)$ be the array obtained by replacing $\mathsf{BWT}[i']$ with a special symbol $\# \notin \Sigma$ if $\mathsf{rank}_{\mathsf{BWT}[i']}(\mathsf{BWT}, i')$ is not a multiple of b. [3] From the definition of BWT'

[3] Here we assume that 0 is not a multiple of b.

we have, for any $i \in [0..n)$ and $c \in \Sigma$,

$$\mathsf{rank}_c(\mathsf{BWT}, i) - b \leq b \cdot \mathsf{rank}_c(\mathsf{BWT}', i) < \mathsf{rank}_c(\mathsf{BWT}, i).$$

Thus, we can obtain $\mathsf{ISA}[j-1]$ from $\mathsf{ISA}[j]$ as follows. First, compute $p := \mathsf{C}[\mathsf{BWT}[i]] + b \cdot \mathsf{rank}_{\mathsf{BWT}[i]}(\mathsf{BWT}', i)$, where $i = \mathsf{ISA}[j]$ and then scan $\mathsf{SA}(p..p+b]$ to find position p' such that $\mathsf{SA}[p'] = j-1$, that is, $\mathsf{ISA}[j-1] = p'$.

Note that we do not need to explicitly store BWT since we can compute it using the suffix array when needed. Sequential scans of SA do not introduce many cache misses and thus are very fast in practice. What remains is to show how to implement rank queries over BWT'.

For each symbol $c \in \Sigma$ we store all occurrences of c in BWT' in a sorted array $\mathsf{S}[c]$, storing at most n/b integers over all symbols. To answer $\mathsf{rank}_c(\mathsf{BWT}', i)$, we binary search the c's list to find the largest i' such that $\mathsf{S}[i'] < i$ and return $i' + 1$ as the result. Thus computing a single LF value using BWT' takes $\mathcal{O}(b + \log(n/b))$ time.

The LZ77 parsing algorithm using the above techniques to reduce the space occupied by ISA runs in $\mathcal{O}(n + z \log n + zkb + zk \log(n/b))$ time and uses $(n+n/k+n/b)\log n+n+\mathcal{O}(\sigma \log n)$ bits of working space. Setting $b = \log n$ and $k = \mathcal{O}(1)$ gives running time $\mathcal{O}(n + z \log n)$ using $(n+n/k)\log n+ 2n + \mathcal{O}(\sigma \log n)$ bits of working space. Given $z = \mathcal{O}(n/\log_\sigma n)$, the time complexity is $\mathcal{O}(n \log \sigma)$. The exact choice of k is the main parameter controlling the time-space tradeoff in the algorithm.

Further space reduction for highly repetitive data. In Paper III we describe an alternative representation of the rank data structure over BWT that is faster and more space efficient if the text is highly repetitive. For such text, the BWT tends to contain long runs of equal symbols [57]. Each run $\mathsf{BWT}[i_1..i_2)$ is encoded using a triple $(i_1, i_2 - i_1, \mathsf{rank}_{\mathsf{BWT}[i_1]}(\mathsf{BWT}, i_1))$. All triples are sorted by the first element and runs of different symbols are stored in separate lists.

To answer $\mathsf{rank}_{\mathsf{BWT}[i]}(\mathsf{BWT}, i)$, we binary search the list of triples encoding runs of $\mathsf{BWT}[i]$ to find the triple (i', ℓ, t) such that $i' \leq i < i' + \ell$ and return $t + i - i'$ as the answer.

Let r be the number of runs in BWT. The above rank representation takes $3r \log n$ bits of space and answers queries in $\mathcal{O}(\log r)$ time. Thus, the parsing algorithms runs in $\mathcal{O}(n + zk \log r + z \log n)$ time and uses $(n + n/k) \log n + 3r \log n + n + \mathcal{O}(\sigma \log n)$ bits of working space. Compared to previous approach, the space usage is reduced if r is significantly smaller than n.

5.4 Scan-based algorithm

All algorithms described so far in this chapter require at least $n \log n$ bits of working space. This limits their scalability to texts that are a few times smaller than the available RAM. To process larger inputs we either need more space-efficient algorithms or we need to resolve to using external memory. In this section we focus on the first option, namely, using much less than $n \log n$ bits of working space. This may be desirable, if the external memory is slow or very limited. Algorithms using external memory are the topic of the next chapter.

We propose an algorithm called LZscan that, for any $d \geq 1$, uses $\mathcal{O}((n \log n)/d)$ bits of working space and runs in $\mathcal{O}(ndt_{\mathrm{rank}})$ time, where $\mathcal{O}(t_{\mathrm{rank}})$ is the time complexity of the rank query. This section is based on Paper IV.

5.4.1 Overview

The general approach of LZscan is similar to the algorithms for suffix and LCP array construction from Chapter 3 and 4. The text is divided into $d = \lceil n/m \rceil$ segments of size m (except possibly the last one) and each of the segments is processed separately. To process a single segment, the algorithm uses $\mathcal{O}(m \log n) = \mathcal{O}((n \log n)/d)$ bits of working space. The segments are processed left-to-right.

Assume that we are currently processing segment $\mathsf{X} = \mathsf{T}[s..e)$ and let W be the concatenation of all segments to the left of X. Our goal is to compute $\mathsf{LPF}_{\mathsf{T}}[s..e)$. For simplicity assume that no phrase or its source crosses segment boundaries. Then the parsing of X can be easily obtained with a single scan over $\mathsf{LPF}_{\mathsf{T}}[s..e)$. The details of handling phrases crossing segment boundaries can be found in Paper IV and are omitted here.

Every $\mathsf{LPF}_{\mathsf{T}}[j]$, $j \in [s..e)$ can be classified into one of two types, depending on whether $p_j \geq s$ (type I) or $p_j < s$ (type II). Type-I LPFs can be easily computed from X in $\mathcal{O}(m)$ time and space using one of many linear-time LPF array construction algorithms [17]. To compute type-II LPFs we first need to introduce a new concept.

Definition 5.2. Given two strings X and W, we define the *matching statistics* of X with respect to W as an array $\mathsf{MS}_{\mathsf{X:W}}[0..|\mathsf{X}|)$ such that $\mathsf{MS}_{\mathsf{X:W}}[j] = (\widehat{p_j}, \widehat{\ell_j})$, where $\mathsf{X}[j..j + \widehat{\ell_j})$ is the longest substring of X starting at position j that also occurs in W and $\mathsf{X}[j..j + \widehat{\ell_j}) = \mathsf{W}[\widehat{p_j}..\widehat{p_j} + \widehat{\ell_j})$. Note that $\widehat{p_j}$ is not uniquely defined.

It is easy to see that the computation of all type-II LPFs is equivalent to the computation of $\mathsf{MS}_{\mathsf{X:W}}$.

5.4.2 Computing matching statistics in small space

Every efficient algorithm computing $\mathsf{MS}_{\mathsf{X:W}}$ that we are aware of (see, e.g., [68, 3, 12]) starts by building an $\mathcal{O}(|\mathsf{W}|)$-size data structure and thus cannot be applied here, since we want to use only $|\mathsf{X}| \log n$ bits of space and it is possible that $|\mathsf{W}| \gg |\mathsf{X}|$. Our solution is to first compute $\mathsf{MS}_{\mathsf{W:X}}$. This only involves linear-size data structures on X which we can afford. Then, we *invert* $\mathsf{MS}_{\mathsf{W:X}}$ into $\mathsf{MS}_{\mathsf{X:W}}$.

Inverting matching statistics. The matching statistic inversion is a novel procedure described in Paper IV that allows efficient computation of $\mathsf{MS}_{\mathsf{X:W}}$ using $\mathcal{O}(|\mathsf{X}| \log n)$ bits of space. The pseudo-code of the inversion algorithm is given in Paper IV (Fig. 1). The data structures required during inversion are SA_{X} and $\mathsf{LCP}_{\mathsf{X}}$ which both take $\mathcal{O}(|\mathsf{X}| \log n)$ bits of space. The procedure also needs $\mathsf{MS}_{\mathsf{W:X}}$, but every element is accessed only once and thus does not need to be stored. The computation of $\mathsf{MS}_{\mathsf{X:W}}$ is therefore reduced to enumerating all values of $\mathsf{MS}_{\mathsf{W:X}}$.

Computing $\mathsf{MS}_{\mathsf{W:X}}$. A standard approach for computing matching statistics [68] when $|\mathsf{W}| \gg |\mathsf{X}|$ works as follows. We scan W right-to-left and for any $j = |\mathsf{W}| - 1, \ldots, 0$, we compute a pair $(\mathsf{P}_j, \widehat{\ell_j})$, where P_j is the interval of positions $\mathsf{P}_j = [s_j..e_j)$ such that $\mathsf{SA}_{\mathsf{X}}[s_j..e_j)$ contains all suffixes of X prefixed with $\mathsf{W}[j..j + \widehat{\ell_j})$.[4] Thus $\mathsf{MS}_{\mathsf{W:X}}[j] = (p, \widehat{\ell_j})$, where p is any value in $\mathsf{SA}_{\mathsf{X}}[s_j..e_j)$. Computing $(\mathsf{P}_j, \widehat{\ell_j})$ requires only access to $\mathsf{W}[j]$, $(\mathsf{P}_{j+1}, \widehat{\ell_{j+1}})$ and some indexing data structures on X. Among others, these data structures include $\mathsf{BWT}_{\mathsf{X}}$ that was preprocessed for rank queries. The time to compute $(\mathsf{P}_j, \widehat{\ell_j})$ is dominated by the rank query on $\mathsf{BWT}_{\mathsf{X}}$. Thus, if by $\mathcal{O}(t_{\text{rank}})$ we denote the time of a single rank query, the whole procedure runs in $\mathcal{O}(|\mathsf{W}| t_{\text{rank}})$ time. Over all segments this amounts to $\mathcal{O}(ndt_{\text{rank}})$ operations.

LZscan uses a slightly modified approach to compute the matching statistics, which we found to be more effective in practice. The scan is performed right-to-left, similar to [68] but instead of maintaining a range P_j, we only keep a single element $p \in \mathsf{P}_j$. For the purpose of matching statistics inversion, this is sufficient. Intuitively, the new method is faster in practice, because rather than updating two endpoints of the range, we

[4] In fact, in [68] the matching statistics are defined as $\mathsf{MS}_{\mathsf{W:X}}[j] = (\mathsf{P}_j, \widehat{\ell_j})$

just update a single element, which reduces the number of rank queries by a factor of two.

Skipping repetitions. Segments are processed left-to-right, thus, when we begin processing X, the factorization of W has already been computed. Knowing the repetition structure of W can sometimes be used to skip the computation of $(\mathsf{P}_j, \widehat{\ell_j})$ for a contiguous range of positions. Suppose $\mathsf{W}[i..i+\ell)$ is an LZ-factor. Thus, there must be another occurrence of $\mathsf{W}[i..i+\ell)$ starting at some position $i' < i$. If a source of any phrase in the parsing of X is completely contained in $\mathsf{W}[i..i+\ell)$, it can be replaced with the equivalent source contained in $\mathsf{W}[i'..i'+\ell)$. Thus, if at any point during the computation of matching statistics we find $\mathsf{MS}_{\mathsf{W:X}}[j] = (p, \widehat{\ell_j})$ that satisfies $i \leq j < j + \widehat{\ell_j} \leq i + \ell$, we can ignore $\mathsf{MS}_{\mathsf{W:X}}[j]$ in the matching statistics inversion. Furthermore, for any $j' \in [i..j)$, we must also have $i \leq j' + \widehat{\ell_{j'}} \leq i + \ell$, hence we can skip all such j' and resume the computation at position $i-1$. This saves time but prevents the computation of $\mathsf{MS}_{\mathsf{W:X}}[i-1]$ using the standard method, since it requires $\mathsf{MS}_{\mathsf{W:X}}[i]$. In this case, we compute $\mathsf{MS}_{\mathsf{W:X}}[i-1]$ using string binary search over the suffix array of X. The string binary search is slower than the normal computation of $\mathsf{MS}_{\mathsf{W:X}}[i-1]$, so we only use this technique, if $\widehat{\ell_j} \geq t$, where t is some predefined threshold. We found $t = 40$ to give the best performance in our experiments.

5.5 Practical performance

Paper III contains an experimental comparison of the new "large space" (i.e., using $\Omega(n \log n)$ bits of working space) algorithms presented in this section to all relevant prior work.

The new algorithm for precomputing the NSV/PSV arrays using $3n \log n$ bits of working space results in the fastest linear-time algorithm for LZ77 parsing. Its optimized version, using $2n \log n$ bits of space achieves a very similar performance. Furthermore, the new algorithms for computing NSV/PSV can be easily modified to solve more general problem, namely computing all LPF values. In our experiments, the new algorithms are the fastest way to obtain the full LPF array.

In the comparison of large space algorithms we also included the new family of algorithms that use data structures capable of answering NSV/PSV queries rather than precomputing all NSV/PSV values. The basic version of the algorithm using a little over $2n \log n$ bits of space achieves a very similar performance to linear-time algorithms using $2n \log n$ bits of space. The space-efficient version of the algorithm allows using much less than $2n \log n$

bits of space ($1.25n \log n$ in our experiments) and is at most 2 times slower than the basic version. For non-repetitive data, this is many times faster than the previously best algorithm at this memory level.

In Paper IV we compared the LZscan algorithm to other lightweight algorithms, all based on compressed text indexes [53, 67]. In our experiments, LZscan achieves the best time-space tradeoff of all algorithms although not by a significant margin. The algorithm, however, notably dominates other solutions on highly repetitive data. This is mainly due to the technique that reuses the already computed part of factorization to partially skip the computation of matching statistics.

Chapter 6

LZ77 parsing in external memory

In this chapter we investigate how to compute the LZ77 parsing for inputs that exceed the size of internal memory. This is particularly important for large highly repetitive collections, which do not fit in RAM, but the resulting LZ77 parsing (and the index built upon it) does.

Some of the new algorithms in this chapter rely on the prior computation of suffix or LCP array. In particular, the most scalable of the algorithms (EM-LPF) requires both arrays. Thus, this chapter additionally motivates the findings from previous chapters.

To the best of our knowledge, we are the first to directly address the problem of external-memory LZ77 parsing. The material in this chapter is based on Paper V. As an extension of the paper, we describe a modification of the EM-LPF algorithm that reduces the disk space usage, and revise the experiments to include the recently published algorithms.

6.1 LPF-based algorithm

Similar to Chapter 5, our first attempt to compute the LZ77 parsing is to compute (and store to disk) the full LPF array of the input string. Then the parsing is easily obtained with a single scan of the LPF array.

Computing the LPF array. Our external memory algorithm constructing the LFP array, called EM-LPF, is an adaptation of the internal-memory LPF computation of Crochemore, Ilie and Smyth [18].

The algorithm takes the suffix and LCP array of the text as input. In Paper V we used eSAIS to compute both arrays, since it was the fastest algorithm that could compute the LCP array for inputs that are larger than the available RAM. In this section we present the extended set of

```
Algorithm EM-LPF
1:  S ← {(−1, −1)}
2:  for i ← 0 to n do
3:      if i = n then (j, ℓ) ← (n, 0)      // empties the stack
4:      else (j, ℓ) ← (SA[i], LCP[i])
5:      while S ≠ ∅ do
6:          (j_s, ℓ_s) ← top(S)
7:          if j < j_s then
8:              pop(S)
9:              if ℓ < ℓ_s then
10:                 (j′, ·) ← top(S)
11:                 LPF[j_s] ← (j′, ℓ_s)
12:             else
13:                 LPF[j_s] ← (j, ℓ)
14:                 ℓ ← ℓ_s
15:         else if j > j_s and ℓ_s ≥ ℓ then
16:             pop(S)
17:             (j′, ·) ← top(S)
18:             LPF[j_s] ← (j′, ℓ_s)
19:         else break
20:     push(S, (j, ℓ))
21: return LPF[0..n)
```

Figure 6.1: LPF array construction in external memory.

experiments, including the algorithms published in the meantime, namely, we consider the combination pSAscan + LCPscan from Chapters 3 and 4 as a replacement of eSAIS.

The algorithm is given in Figure 6.1. For clarity, we present the internal-memory version and explain how to implement it in external memory. Compared to the original algorithm of Crochemore et al. [18], we made the following modifications:

- The algorithm computes the LPF array, as defined in Chapter 5 (i.e., as a sequence of pairs encoding both the length and the position of the previous occurrence), which is necessary for the purpose of computing the Lempel-Ziv parsing. The original algorithm only computes the length-component.

- The suffix array value $\mathsf{SA}[i]$ in the original algorithm is encoded on stack by index i. We store the actual value $\mathsf{SA}[i]$ instead and thus avoid random access to SA.

- The original algorithm in some cases modifies the LCP array. In our

version, every value read from the LCP array is copied (and modified, if necessary) onto stack and from then on it is always accessed from there.

As seen in Figure 6.1 the resulting algorithm only requires sequential access to SA and LCP, thus we can stream those arrays from disk using two small buffers.

The only non-sequential operations are assignments $\mathsf{LPF}[j_s] \leftarrow (j, \ell)$ in lines 11, 13, and 18. However, each position in LPF is assigned a value only once. We can therefore implement these assignments in external memory by sorting the triples (j_s, j, ℓ) by the first component. The EM sorting is performed with the STXXL library [19].

Finally, the algorithm requires some space for the stack. Crochemore et al. [18] proved that at any point during the algorithm, the stack size is bounded by $\mathcal{O}(\sqrt{n})$. For all inputs we tried in experiments, the stack never grows beyond a negligible size. However, to prevent the worst case, we use an external-memory stack from the STXXL library, which only keeps the top $\mathcal{O}(B)$ items in RAM and the rest is stored on disk.

Correctness. Let $\mathsf{LPF}[j] = (p_j, \ell_j)$. As shown by Lemma 5.1, to find p_j it suffices to consider two positions in T, namely: $\mathsf{PSV}_{\mathsf{text}}[j]$ and $\mathsf{NSV}_{\mathsf{text}}[j]$. These positions, however, are not the only valid candidates for p_j. For $i \in [0..n)$, let

$$\mathsf{PSV}'_{\mathsf{lex}}[i] = \min\{i' \in [0..i) \mid \mathsf{SA}[i'] < \mathsf{SA}[i] \text{ and } \mathsf{lcp}(\mathsf{SA}[i], \mathsf{SA}[i']) = \mathsf{lcp}(\mathsf{SA}[i], \mathsf{SA}[\mathsf{PSV}_{\mathsf{lex}}[i]])\}.$$

As before, we also define

$$\mathsf{PSV}'_{\mathsf{text}}[j] = \mathsf{SA}[\mathsf{PSV}'_{\mathsf{lex}}[\mathsf{ISA}[j]]].$$

It is easy to see from the definition that $\mathsf{PSV}'_{\mathsf{text}}[j]$ can replace $\mathsf{PSV}_{\mathsf{text}}[j]$ as a candidate for p_j. The correctness of EM-LPF is based on the following invariant: after processing the pair $(\mathsf{SA}[i], \mathsf{LCP}[i])$, the content of the stack (top to bottom) is: $(\mathsf{SA}[i], \cdot)$, $(\mathsf{PSV}'_{\mathsf{text}}[\mathsf{SA}[i]], \cdot)$, $(\mathsf{PSV}'_{\mathsf{text}}[\mathsf{PSV}'_{\mathsf{text}}[\mathsf{SA}[i]]], \cdot)$, $\ldots$, $(-1, \cdot)$. Note that it is the same as in the internal-memory algorithm from Section 5.2, except $\mathsf{PSV}_{\mathsf{text}}$ has been replaced with $\mathsf{PSV}'_{\mathsf{text}}$. The second component of each pair (p, ℓ) on stack is the length of lcp between suffix p and the next suffix on the stack ($\mathsf{PSV}'_{\mathsf{text}}[p]$).

I/O complexity. The I/O complexity of the LPF array construction is dominated by the external-memory sorting of n triples of integers and therefore is equal to $\mathcal{O}(\mathrm{sort}(n))$. If the SA and LCP array is constructed using eSAIS or another algorithm with sorting complexity, the I/O complexity of the whole algorithm stays the same. Replacing eSAIS with the combination (p)SAscan + LCPscan increases the I/O complexity to that of (p)SAscan (see Chapter 3).

Reducing the disk space usage. Each element of the triple (j_s, j, ℓ) processed by the algorithm in Figure 6.1 needs $\log n$ bits of space. The algorithm sorts n triples, thus the disk space required by the external-memory sorting is $3n \log n$ bits. In some applications, this can be prohibitively high. Depending on the used suffix and LCP array construction algorithms, it can even dominate the peak disk space usage of the whole algorithm, e.g., this is the case when using the combination (p)SAscan + LCPscan.

The disk space usage of the external-memory sorting can be reduced using a similar technique as in Chapter 4. We divide the LPF array into q equal-sized parts and run q rounds of the algorithm. In each round we compute one part of the LPF array. More precisely, the triples computed in lines 11, 13, and 18 are only included in the sorting, if j_s belongs to the currently processed part. After a single round of processing is finished, we immediately scan the computed part of the LPF array (to obtain LZ-factors) and then discard it, before starting the next round. This way, the disk space usage of the sorting is reduced to $(3n \log n)/q$ bits.

A single round of processing requires scanning the whole SA and LCP array. With q parts the I/O volume is increased by $2(q-1)n \log n$ bits.

The above technique was not described in Paper V, thus the section on practical performance in this chapter contains a small experiment showing its effect on the runtime.

6.2 External memory LZscan

A different approach to LZ77 factorization for inputs exceeding the size of RAM is an external-memory variant of the LZscan algorithm described in Chapter 5 that we call EM-LZscan. The key differences compared to the internal-memory version are:

- The text is stored and accessed from disk, rather than internal memory. This is possible, since the single step of matching statistics computation requires only access to a single character and all characters are accessed sequentially.

- All of RAM is designated for the data structures used during the computation of $\mathsf{MS}_{\mathsf{W:X}}$ and the matching statistics inversion. These data structures take $\mathcal{O}(m)$ words, and so we use $m = \Theta(M)$.

Assuming that each symbol needs $\log \sigma$ bits, computing $\mathsf{MS}_{\mathsf{W:X}}$ takes $(|\mathsf{W}| \log \sigma)/(B \log n)$ I/Os. Given $m = \Theta(M)$, the text is divided into $\Theta(n/M)$ segments. The total I/O complexity of the algorithm is therefore

$$\mathcal{O}\left(\frac{n^2 \log \sigma}{MB \log n}\right).$$

The CPU complexity becomes

$$\mathcal{O}\left(\frac{n^2 \cdot t_{\text{rank}}}{M}\right).$$

6.3 Semi-external LZ77 parsing

Our third algorithm computing the LZ77 parsing is semi-external, that is, requires the input text to fit completely in RAM, but other data is stored on disk. The algorithm, called SE-KKP, is based on the internal-memory LZ77 parsing algorithm that precomputes all $\mathsf{NSV}_{\mathsf{text}}/\mathsf{PSV}_{\mathsf{text}}$ values (see Section 5.2). As before, the algorithm takes the input text T and the suffix array of T as input. The main change is that the NSV/PSV arrays are now stored on disk. After the computation of NSV/PSV values is completed, the LZ77 parsing is obtained using the algorithm in Figure 5.1. It does a single sequential scan of $\mathsf{NSV}_{\mathsf{text}}/\mathsf{PSV}_{\mathsf{text}}$, but requires random access to the text, which therefore has to be kept in RAM.

The pseudo-code of the algorithm is given in Figure 6.2. Compared to the original internal memory algorithm from Section 5.2, we need the following changes:

- The stack no longer overwrites the SA. Instead we maintain a separate stack. To guarantee a small memory footprint, we use an external-memory stack from the STXXL library.

- Both $\mathsf{NSV}_{\mathsf{text}}$ and $\mathsf{PSV}_{\mathsf{text}}$ values are always computed and accessed together, hence the arrays are stored interleaved in an array denoted $\mathsf{NPSV}_{\mathsf{text}}$. This simplifies the external memory sorting and reduces the disk seek time, when scanning $\mathsf{NSV}_{\mathsf{text}}/\mathsf{PSV}_{\mathsf{text}}$ arrays in the final step.

Algorithm SE-KKP
1: $\mathcal{S} \leftarrow \{-1\}$
2: **for** $i \leftarrow 0$ **to** n **do**
3: **if** $i = n$ **then** $j \leftarrow -1$ // empties the stack
4: **else** $j \leftarrow \mathsf{SA}[i]$
5: $j_s \leftarrow \mathsf{top}(\mathcal{S})$
6: **while** $j_s > j$ **do**
7: $\mathsf{pop}(\mathcal{S})$
8: $\mathsf{NPSV}_{\mathsf{text}}[j_s] \leftarrow (j, \mathsf{top}(\mathcal{S}))$
9: $j_s \leftarrow \mathsf{top}(\mathcal{S})$
10: $\mathsf{push}(\mathcal{S}, j)$
11: **return** $\mathsf{NPSV}_{\mathsf{text}}[0..n)$

Figure 6.2: Construction of the $\mathsf{NSV}_{\mathsf{text}}/\mathsf{PSV}_{\mathsf{text}}$ arrays (stored interleaved as a single array $\mathsf{NPSV}_{\mathsf{text}}$) in external memory.

The only non-sequential memory access in the algorithm is the assignment $\mathsf{NPSV}_{\mathsf{text}}[j_s] \leftarrow (j, \mathsf{top}(\mathcal{S}))$ in line 8. The solution is analogous to EM-LPF: external memory sort of triples $(j_s,\ j,\ \mathsf{top}(\mathcal{S}))$ by the first component. As before, the sorting is accomplished using the STXXL library.

SA construction. The remaining problem is the computation of SA. Any external memory algorithms, such as eSAIS [10], could be used. However, in this scenario, using an asymptotically slower algorithm such as SAscan (see Chapter 3) could be a better choice. Since the RAM has to be large enough to accommodate the text, we have $n/M \leq \log_\sigma n$, e.g., for byte alphabet and inputs up to 1 TiB, the ratio n/M never exceeds five. For such ratios SAscan achieves comparable performance to eSAIS (possibly even better, if we use its parallel version [43]) and uses much less disk space.

I/O complexity. Sorting the triples in external memory takes $\mathcal{O}(\mathrm{sort}(n))$ $= \mathcal{O}((n/B)\log_{M/B}(n/B))$ I/Os, which simplifies to $\mathcal{O}(n/B)$, assuming $M \log n \geq n \log \sigma$ (see above). The complexity of suffix sorting under this assumption is also $\mathcal{O}(n/B)$ for both eSAIS and (p)SAscan.

6.4 Practical performance

We now present an experimental evaluation of the algorithms described in this chapter. For experiments we used a machine equipped with two 1.9

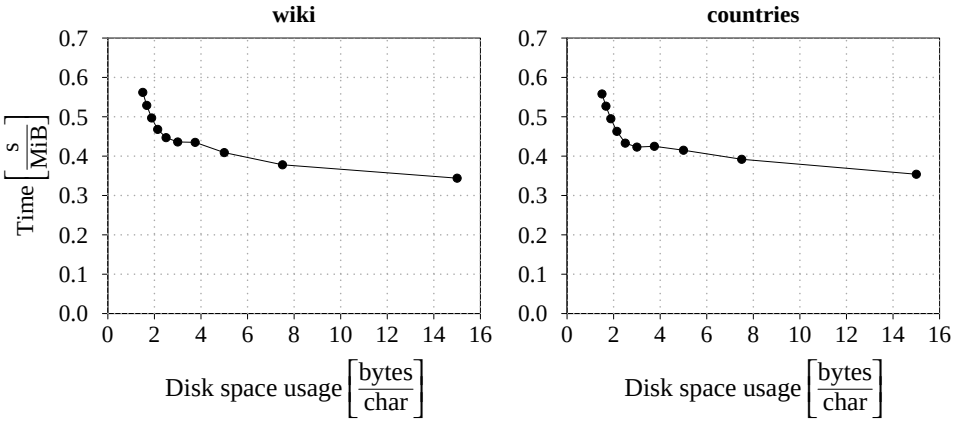


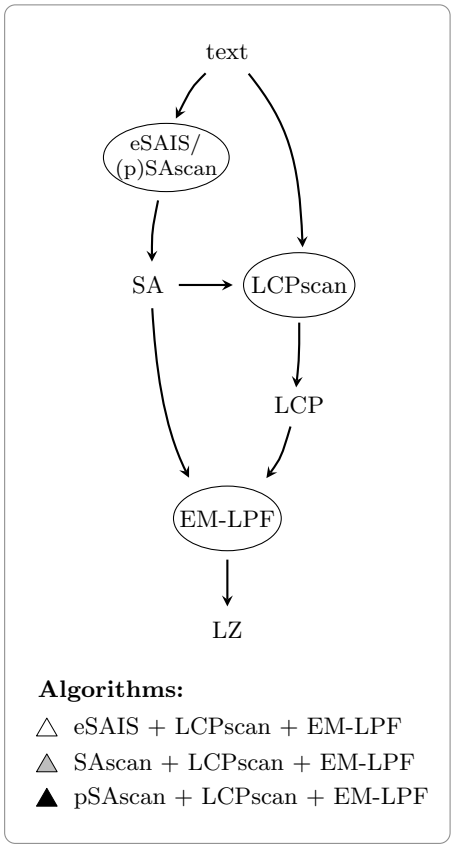
Figure 6.3: Runtime and disk space usage of EM-LPF with varying number of parts (1–10). Both testfiles are of size 32 GiB.

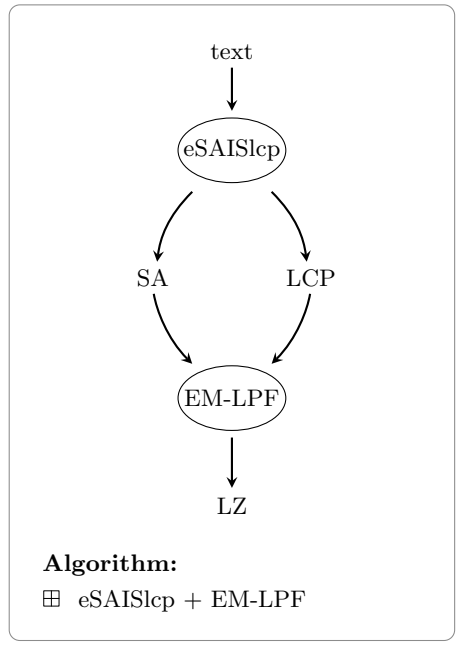
GHz Intel Xeon E5-2420 CPUs with 12 cores in total, 120 GiB of RAM, and 7.2 TiB of disk space striped with RAID0 across 4 local disks of size 1.8 TiB achieving a combined transfer rate of about 480 MiB/s. For experiments we restricted the RAM size to 4 GiB, and the algorithms were allowed to use at most 3.5 GiB. The OS was Linux (Ubuntu 12.04, 64bit). All programs were compiled using `g++` version 4.7.3 with `-O3 -DNDEBUG` options. The pSAscan implementation uses the full parallelism on the machine. In experiments we used two 32 GiB files:

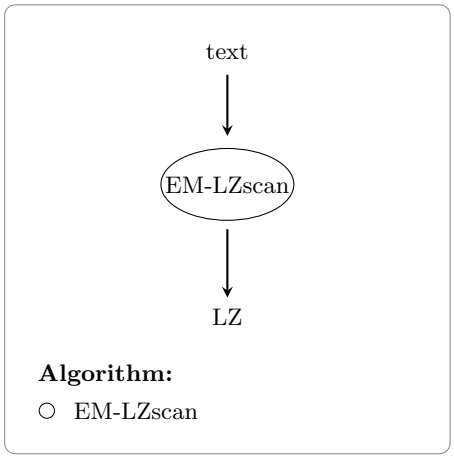
- wiki: a prefix of English Wikipedia dump (dated 20140707) in the XML format,
- countries: a concatenation of all versions (editing history) of four Wikipedia articles about countries in the XML format. It contains a large number of 1–5 KiB repetitions.

First, we investigate the optimization of EM-LPF reducing the disk space usage. We run EM-LPF using the number of parts varying from 1 to 10 and measured the runtime and disk space usage. The results are presented in Figure 6.3.

The repetitiveness of the input data does not have a significant effect on the runtime. As the number of parts is increased, the algorithm gradually slows down due to additional scans of SA. However, the slowdown is very moderate, especially compared to the disk space reduction. With ten parts, the runtime is increased by 64% (compared to the computation with one part), but the working space of the algorithm (excludes input and output) is reduced from $15n$ bytes to only $1.5n$ bytes.







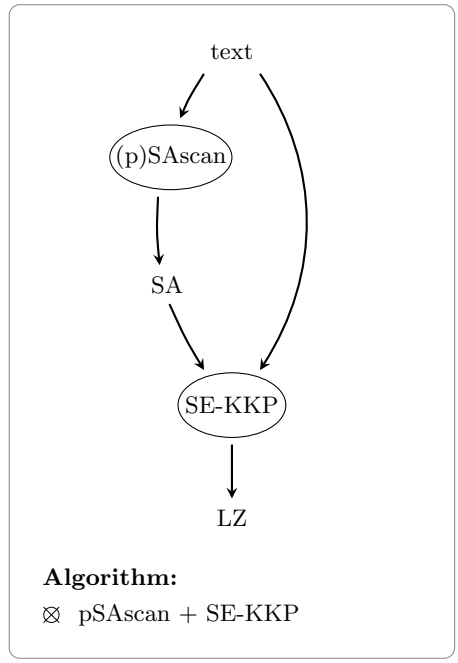


Figure 6.4: An overview of different approaches to computing LZ77 parsing in external memory.

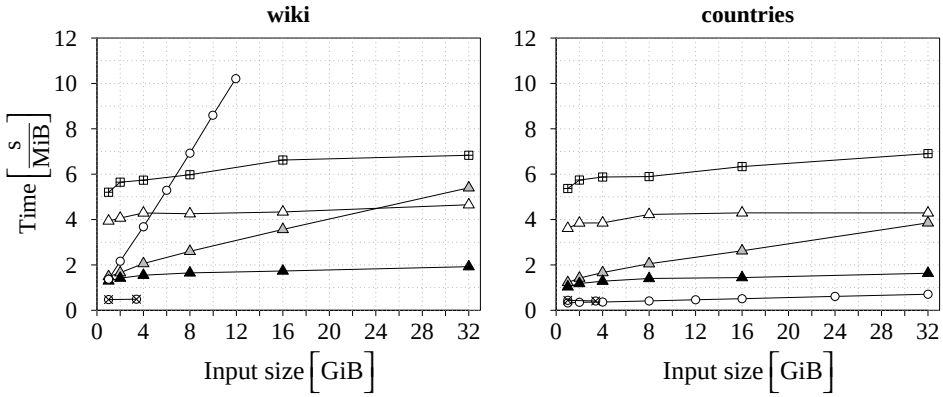


Figure 6.5: Runtime comparison of algorithms computing LZ77 parsing in external memory.

Algorithm	Disk space
eSAIS	$28n$
eSAISlcp	$54n$
(p)SAscan	$6.5n$[1]
LCPscan ($q = 10$)	$12n$
EM-LPF ($q = 10$)	$12.5n$
SE-KKP	$21n$
EM-LZscan	$1.5n$

Table 6.1: Peak disk space usage (in bytes) of all algorithms used in experiments.

In the second experiment, we compare the scalability of all approaches for computing the LZ77 parsing in external memory. This is a revised version of the experiment from Paper V. We additionally include LCPscan (see Chapter 4), the space-efficient version of EM-LPF described in this chapter, and pSAscan [43] – the parallel version of SAscan.

Figure 6.4 contains an overview of all algorithms included in the comparison. By eSAIS we denote a version of the algorithm that only computes the suffix array. A version computing suffix and LCP array is denoted eSAISlcp. For both LCPscan and EM-LPF we used space-efficient implementations, which perform the computation in $q = 10$ parts. Table 6.1 summarizes the peak disk space usage (excluding the varying size output for the algorithms that compute the LZ77 parsing) of all algorithms, assuming 40-bit integers

[1]Assuming that the segment size m satisfies $m \leq 2^{32}$. If we allow $m \leq 2^{40}$, the disk space usage increases to $7.5n$ bytes

and 8-bit text characters.

As seen from the results in Figure 6.5, the best solution varies, depending on the size of input and the degree of repetitiveness. For highly repetitive data (countries), the fastest algorithm is EM-LZscan – this is mostly due to the technique that reuses the already computed part of factorization to skip the computation of matching statistics (Section 5.4.2).

For non-repetitive input (wiki), EM-LZscan is outperformed by other algorithms. The fastest solution in general is a combination pSAscan + LCPscan + EM-LPF. For inputs not exceeding the size of RAM, the combination pSAscan + SE-KKP runs about three times faster.

If the machine does not have many cores (and thus using a parallel version of SAscan does not improve the runtime), the best solution would be the combination SAscan + LCPscan + EM-LPF though only up to a point, where the input is 6.8–10 times larger than the size of RAM. For larger inputs, replacing SAscan with eSAIS gives a better performance, although the disk space usage increases significantly.

Chapter 7

Conclusions

We now present an overview of the thesis. For each chapter we give a short summary of the results and a discussion of open problems.

Chapter 3. We described a series of improvements to the suffix array construction algorithm of Ferragina et al. [22]. The resulting algorithm uses very little extra disk space and is faster than previous suffix-sorting algorithms up to a point where the ratio between the length of the input text and the RAM size is about five. Our follow-up work on parallelizing the algorithm moves this point to about 80 times the size of RAM [43].

Many avenues for future work remain. First, it is possible that a similar speedup from multiple processing units can be obtained on other platforms, such as GPU. Furthermore, the modular structure of the algorithm (i.e., composing the suffix array from many, almost independently constructed partial suffix arrays) could allow a distributed implementation.

Another potential improvement is exploiting the repetitions in the input text to save the computation, similar to how it has been done with LZ77 factorization (see Chapters 5 and 6).

Finally, replacing the current rank data structure with some compressed representation that takes less space but is not much slower would allow using larger segments and thus improve the overall performance of the algorithm. Recently, Tischler [77] proposed a semi-external algorithm for suffix array construction that uses the same general approach as SAscan but also heavily utilizes compressed representations (in particular for rank). The algorithm has not been implemented yet, thus it remains to be seen if this approach will work well in practice.

Chapter 4. We proposed the first algorithm computing the LCP array in external memory that is not an extension of suffix array construction. Despite having worse I/O complexity than the best external memory suffix array construction algorithms [10, 47], the new algorithm is fast in practice and uses very little disk space in addition to input and output.

The running time of LCPscan in our experiments is dominated by the computation (not I/O) performed by the STXXL library during the external-memory sort. Thus, the first avenue for future work is optimizing the STXXL sorting. The scanning/sorting nature of the algorithm suggests that it may also be possible to implement LCPscan in a distributed setting.

The I/O volume of LCPscan is about $100n$, whereas the I/O volume of the best semi-external LCP array construction algorithms (that require the text to fit in RAM) is about $16n$ [44]. Thus, there is a big gap between the two and the transition between the text that fit in RAM and the text that is just a little too large to fit in RAM is not smooth. We are currently working on addressing this issue by designing a new semi-external algorithm that can handle texts slightly larger than RAM much faster than LCPscan.

Finally, it remains an open problem, whether the LCP array construction in external memory can match the I/O complexity of sorting n integers $\mathcal{O}(\mathrm{sort}(n))$, without being an extension of the suffix array construction algorithm. We are currently working on an algorithm with $\mathcal{O}(\mathrm{sort}(n) \log \sigma)$ I/O complexity.

Chapter 5. We described three new algorithms (some of which have many variants) for computing the LZ77 parsing in RAM, spanning a wide range in the time/space spectrum. The new algorithms consistently outperform prior methods or use less space or both. As shown in Chapter 6, some of the new methods generalize well to external memory.

One of the problems that remains unsolved is whether the LZ77 parsing can be computed in linear time using $n \log n$ bits of working space for an integer alphabet. By a suitable choice of parameters the algorithm based on NSV/PSV queries described in Chapter 5 achieves $(1+\epsilon)n \log n + n + \mathcal{O}(\sigma \log n)$ working space and runs in $\mathcal{O}((n \log \sigma)/\epsilon)$ time, where $\epsilon > 0$ is a fixed constant. The linear-time algorithm of Goto and Bannai [38] uses $n \log n + \mathcal{O}(\sigma \log n)$ bits of space, thus improving upon our result, but still requires small σ. The closest to the final solution is the recent linear-time algorithm of Fischer et al. [28]. Their algorithm works for $\sigma = n^{\mathcal{O}(1)}$ and uses $(1+\epsilon)n \log n + \mathcal{O}(n)$ bits of working space which, unlike other algorithms, includes also the space for output.

Another avenue for future work is parallelism. The only existing parallel

implementation of LZ77 factorization that we are aware of is due to Shun and Zhao [73]. It achieves a good speedup but suffers from high memory consumption. The LZscan algorithm has a similar structure to SAscan, which was shown to greatly benefit from parallelism [43], thus a parallel version of LZscan could also achieve a good speedup.

Finally, since the fastest algorithms for LZ parsing in our experiments in Paper III depend on the degree of repetition (there are at least three different algorithms that stand out), an interesting problem is estimating the size of LZ77 parsing. Such a tool could guide the use of the best algorithm. We are aware of two papers on this topic [27, 29], although none of the algorithms have been implemented yet.

Chapter 6. We proposed three new algorithms for computing the LZ77 parsing in external memory. To the best of our knowledge, we are the first to describe algorithms for this problem. In our experiments, the new algorithms scale very well with the input and require moderate disk space.

As demonstrated by the experiments, there is a large discrepancy in the efficiency of EM-LZscan between regular and highly-repetitive input. Thus, the algorithms estimating the size of LZ77 parsing are not only needed in internal memory (see above) but should also scale to external memory.

Finally, the problem that has received very little attention is LZ77 decoding. It has a simple linear-time solution in internal memory and runs very fast in practice. However, when moving into external memory, the problem becomes non-trivial due to lack of locality of LZ77 phrase sources.

Our implementations of all of the algorithms introduced in this thesis are publicly available [1] and thus directly accessible for practitioners. This will also allow the code to be customized and further developed. We hope that our implementations (especially for the problems that were addressed for the first time) will help to establish a baseline for researchers to benchmark their algorithms and implementations in the future.

We speculate that with the rapid development of sequencing machines and molecular biology in general, the line of research pursued in this thesis will keep gaining the attention and the need for practical algorithms processing large amounts of textual data will grow. One of the commonly anticipated scenarios involves increased use of distributed/cloud computing for which many of our new algorithms are likely to adapt.

[1] `http://www.cs.helsinki.fi/group/pads/`

References

[1] Strategy for UK life sciences: One year on. `https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/36684/12-1346-strategy-for-uk-life-sciences-one-year-on.pdf`, 2012.

[2] A. Abeliuk, R. Cánovas, and G. Navarro. Practical compressed suffix trees. *Algorithms*, 6(2):319–351, 2013.

[3] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004.

[4] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[5] A. Al-Hafeedh, M. Crochemore, L. Ilie, E. Kopylov, W. F. Smyth, G. Tischler, and M. Yusufu. A comparison of index-based Lempel-Ziv LZ77 factorization algorithms. *ACM Comput. Surv.*, 45(1):5, 2012.

[6] G. Badkobeh, M. Crochemore, and C. Toopsuwan. Computing the maximal-exponent repeats of an overlap-free string in linear time. In *Proceedings of the 19th International Symposium on String Processing and Information Retrieval (SPIRE'12)*, volume 7608 of *LNCS*, pages 61–72. Springer, 2012.

[7] J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet partitioning for compressed rank/select and applications. In *Proceedings of the 21st International Symposium on Algorithms and Computation (ISAAC'10)*, volume 6507 of *LNCS*, pages 315–326. Springer, 2010.

[8] D. Belazzougui and S. J. Puglisi. Range predecessor and Lempel-Ziv parsing. *CoRR*, abs/1507.07080, 2015.

[9] T. Beller, S. Gog, E. Ohlebusch, and T. Schnattinger. Computing the longest common prefix array based on the Burrows-Wheeler transform. *J. Discrete Algorithms*, 18:22–31, 2013.

[10] T. Bingmann, J. Fischer, and V. Osipov. Inducing suffix and LCP arrays in external memory. In *Proceedings of the 2013 Workshop on Algorithm Engineering and Experiments (ALENEX'13)*, pages 88–102. SIAM, 2013.

[11] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[12] W. I. Chang and E. L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4–5):327–344, 1994.

[13] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Rasala, A. Sahai, and a. shelat. Approximating the smallest grammar: Kolmogorov complexity in natural models. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC'02)*, pages 792–801. ACM, 2002.

[14] G. Chen, S. J. Puglisi, and W. F. Smyth. Lempel-Ziv factorization using less time and space. *Math. Comput. Sci.*, 1(4):605–623, 2008.

[15] A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[16] M. Crochemore and L. Ilie. Computing longest previous factor in linear time and applications. *Inform. Process. Lett.*, 106(2):75–80, 2008.

[17] M. Crochemore, L. Ilie, C. S. Iliopoulos, M. Kubica, W. Rytter, and T. Walen. LPF computation revisited. In *Proceedings of the 20th International Workshop on Combinatorial Algorithms (IWOCA'09)*, volume 5874 of *LNCS*, pages 158–169. Springer, 2009.

[18] M. Crochemore, L. Ilie, and W. F. Smyth. A simple algorithm for computing the Lempel-Ziv factorization. In *Proceedings of the 2008 Data Compression Conference (DCC'08)*, pages 482–488. IEEE Computer Society, 2008.

[19] R. Dementiev, L. Kettner, and P. Sanders. STXXL: standard template library for XXL data sets. *Softw., Pract. Exper.*, 38(6):589–637, 2008.

[20] R. Durbin et al. 1000 genomes. `http://www.1000genomes.org/`, 2010.

[21] H. Ferrada, T. Gagie, T. Hirvola, and S. J. Puglisi. Hybrid indexes for repetitive datasets. *Phil. Trans. R. Soc. A*, 372(2016), 2014.

[22] P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.

[23] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS'00)*, pages 390–398. IEEE Computer Society, 2000.

[24] P. Ferragina and G. Manzini. On compressing the textual web. In *Proceedings of the 3rd ACM International Conference on Web Search and Data Mining (WSDM'10)*, pages 391–400. ACM, 2010.

[25] J. Fischer. Combined data structure for previous- and next-smaller-values. *Theor. Comput. Sci.*, 412(22):2451–2456, 2011.

[26] J. Fischer. Inducing the LCP-array. In *Proceedings of the 2011 Algorithms and Data Structures Symposium (WADS'11)*, volume 6844 of *LNCS*, pages 374–385. Springer, 2011.

[27] J. Fischer, T. Gagie, P. Gawrychowski, and T. Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15)*, volume 9294 of *LNCS*, pages 533–544. Springer, 2015.

[28] J. Fischer, T. I, and D. Köppl. Lempel Ziv computation in small space (LZ-CISS). In *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM'15)*, volume 9133 of *LNCS*, pages 172–184. Springer, 2015.

[29] T. Gagie. Approximating LZ77 in small space. *CoRR*, abs/1503.02416, 2015.

[30] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. A faster grammar-based self-index. In *Proceedings of the 6th International Conference on Language and Automata Theory and Applications (LATA'12)*, volume 7183 of *LNCS*, pages 240–251. Springer, 2012.

[31] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. LZ77-based self-indexing with faster pattern matching. In *Proceedings of the 11th Latin American Symposium on Theoretical Informatics (LATIN'14)*, volume 8392 of *LNCS*, pages 731–742. Springer, 2014.

[32] T. Gagie, P. Gawrychowski, and S. J. Puglisi. Faster approximate pattern matching in compressed repetitive texts. In *Proceedings of the 22nd International Symposium on Algorithms and Computation (ISAAC'11)*, volume 7074 of *LNCS*, pages 653–662. Springer, 2011.

[33] Genome 10K Community of Scientists. A proposal to obtain whole-genome sequence for 10,000 vertebrate species. *J. Hered.*, 100:659–674, 2009.

[34] S. Gog. *Compressed Suffix Trees: Design, Construction, and Applications.* PhD thesis, Ulm University, 2011.

[35] S. Gog and E. Ohlebusch. Fast and lightweight LCP-array construction algorithms. In *Proceedings of the 2011 Workshop on Algorithm Engineering and Experiments (ALENEX'11)*, pages 25–34. SIAM, 2011.

[36] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: Pat trees and Pat arrays. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, pages 66–82. Prentice–Hall, 1992.

[37] K. Goto and H. Bannai. Simpler and faster Lempel Ziv factorization. In *Proceedings of the 2013 Data Compression Conference (DCC'13)*, pages 133–142. IEEE Computer Society, 2013.

[38] K. Goto and H. Bannai. Space efficient linear time Lempel-Ziv factorization for small alphabets. In *Proceedings of the 2014 Data Compression Conference (DCC'14)*, pages 163–172. IEEE Computer Society, 2014.

[39] J. Kärkkäinen. *Repetition-Based Text Indexes.* PhD thesis, University of Helsinki, 1999.

[40] J. Kärkkäinen, D. Kempa, and M. Piątkowski. Tighter bounds for the sum of irreducible LCP values. In *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM'15)*, volume 9133 of *LNCS*, pages 316–328. Springer, 2015.

[41] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Linear time Lempel-Ziv factorization: simple, fast, small. In *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM'13)*, volume 7922 of *LNCS*, pages 189–200. Springer, 2013.

[42] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. String range matching. In *Proceedings of the 25th Annual Symposium on Combinatorial Pattern Matching (CPM'14)*, volume 8486 of *LNCS*, pages 232–241. Springer, 2014.

[43] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Parallel external memory suffix sorting. In *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching (CPM'15)*, volume 9133 of *LNCS*, pages 329–342. Springer, 2015.

[44] J. Kärkkäinen, G. Manzini, and S. J. Puglisi. Permuted longest-common-prefix array. In *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM'09)*, volume 5577 of *LNCS*, pages 181–192. Springer, 2009.

[45] J. Kärkkäinen and S. J. Puglisi. Fixed-block compression boosting in FM-indexes. In *Proceedings of the 18th International Symposium on String Processing and Information Retrieval (SPIRE'11)*, volume 7024 of *LNCS*, pages 174–184. Springer, 2011.

[46] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP'03)*, volume 2719 of *LNCS*, pages 943–955. Springer, 2003.

[47] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[48] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01)*, volume 2089 of *LNCS*, pages 181–192. Springer, 2001.

[49] D. Kempa and S. J. Puglisi. Lempel-Ziv factorization: simple, fast, practical. In *Proceedings of the 2013 Workshop on Algorithm Engineering and Experiments (ALENEX'13)*, pages 103–112. SIAM, 2013.

[50] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4):143–156, 2005.

[51] R. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS'99)*, pages 596–604. IEEE Computer Society, 1999.

[52] D. Kosolobov. Faster lightweight Lempel-Ziv parsing. In *Proceedings of the 40th International Symposium on Mathematical Foundations of Computer Science (MFCS'15)*, volume 9235 of *LNCS*, pages 432–444. Springer, 2015.

[53] S. Kreft and G. Navarro. LZ77-like compression with fast random access. In *Proceedings of the 2010 Data Compression Conference (DCC'10)*, pages 239–248. IEEE Computer Society, 2010.

[54] S. Kreft and G. Navarro. Self-indexing based on LZ77. In *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM'11)*, volume 6661 of *LNCS*, pages 41–54. Springer, 2011.

[55] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.*, 483:115–133, 2013.

[56] V. Mäkinen. Compact suffix array — a space efficient full-text index. *Fund. Inform.*, 56(1–2):191–210, 2003.

[57] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comp. Biol.*, 17(3):281–308, 2010.

[58] U. Manber and G. W. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[59] G. Manzini. Two space saving tricks for linear time LCP array computation. In *Proceedings of the 14th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT'04)*, volume 3111 of *LNCS*, pages 372–383. Springer, 2004.

[60] Y. Mori. libdivsufsort, a C library for suffix array construction. `http://code.google.com/p/libdivsufsort/`.

[61] Y. Mori. SAIS, an implementation of the induced sorting algorithm. `https://sites.google.com/site/yuta256/sais`.

[62] G. Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014.

[63] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1):article 2, 2007.

[64] G. Nong, S. Zhang, and W. H. Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011.

[65] E. Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.

[66] E. Ohlebusch, J. Fischer, and S. Gog. CST++. In *Proceedings of the 17th International Symposium on String Processing and Information Retrieval (SPIRE'10)*, volume 6393 of *LNCS*, pages 322–333. Springer, 2010.

[67] E. Ohlebusch and S. Gog. Lempel-Ziv factorization revisited. In *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM'11)*, volume 6661 of *LNCS*, pages 15–26. Springer, 2011.

[68] E. Ohlebusch, S. Gog, and A. Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *Proceedings of the 17th International Symposium on String Processing and Information Retrieval (SPIRE'10)*, volume 6393 of *LNCS*, pages 347–358. Springer, 2010.

[69] D. Okanohara and K. Sadakane. An online algorithm for finding the longest previous factors. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*, volume 5193 of *LNCS*, pages 696–707. Springer, 2008.

[70] I. Pavlov. 7-zip. `http://www.7-zip.org/`, 2012.

[71] S. J. Puglisi and A. Turpin. Space-time tradeoffs for longest-common-prefix array computation. In *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC'08)*, volume 5369 of *LNCS*, pages 124–135. Springer, 2008.

[72] P. Sanders. Algorithm engineering - an attempt at a definition using sorting as an example. In *Proceedings of the 2010 Workshop on Algorithm Engineering and Experiments (ALENEX'10)*, pages 55–61. SIAM, 2010.

[73] J. Shun and F. Zhao. Practical parallel Lempel-Ziv factorization. In *Proceedings of the 2013 Data Compression Conference (DCC'13)*, pages 123–132. IEEE Computer Society, 2013.

[74] J. Sirén. Sampled longest common prefix array. In *Proceedings of the 21st Annual Symposium on Combinatorial Pattern Matching (CPM'10)*, volume 6129 of *LNCS*, pages 227–237. Springer, 2010.

[75] J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE'08)*, volume 5280 of *LNCS*, pages 164–175. Springer, 2008.

[76] T. A. Starikovskaya. Computing Lempel-Ziv factorization online. In *Proceedings of the 37th International Symposium on Mathematical Foundations of Computer Science (MFCS'12)*, volume 7464 of *LNCS*, pages 789–799. Springer, 2012.

[77] G. Tischler. Faster average case low memory semi-external construction of the Burrows–Wheeler transform. In *Proceedings of the 2nd International Conference on Algorithms for Big Data (ICABD'14)*, volume 1146 of *CEUR Workshop Proceedings*, pages 61–68. CEUR-WS.org, 2014.

[78] J. S. Vitter. Algorithms and data structures for external memory. *Found. Trends Theoretical Computer Science*, 2(4):305–474, 2006.

[79] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT'73)*, pages 1–11. IEEE Computer Society, 1973.

[80] H. E. Williams and J. Zobel. Compressing integers for fast file access. *Comput. J.*, 42(3):193–201, 1999.

[81] J. Yamamoto, T. I, H. Bannai, S. Inenaga, and M. Takeda. Faster compact on-line Lempel-Ziv factorization. In *Proceedings of the 31st International Symposium on Theoretical Aspects of Computer Science (STACS'14)*, volume 25 of *LIPIcs*, pages 675–686, 2014.

[82] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.

Symbols and abbreviations

Symbol	Page	Semantics
B	9	Disk block size (in $\log n$-bit words)
BWT	8	Burrows-Wheeler transform
$\mathsf{BWT}_{S:S'}$	15	BWT of string SS' restricted to suffixes starting in S
$\mathsf{C}[c]$	16	Number of occurrences of characters $c' < c$ in the text
GiB	-	Gibibyte (2^{30} bytes)
ISA	7	Inverse of SA
KiB	-	Kibibyte (2^{10} bytes)
LCP	8	Longest common prefix array
LF	38	LF-mapping
$\mathsf{LPF}[i]$	8	Longest previous factor at position i
M	9	Size of RAM (in $\log n$-bit words)
MiB	-	Mebibyte (2^{20} bytes)
$\mathsf{MS}_{S:S'}$	40	Matching statistics of string S wrt to string S'
$\mathsf{NSV}_{\mathsf{lex}}$	34	NSV array in lexicographical order
$\mathsf{NSV}_{\mathsf{text}}$	35	NSV array in text order
PLCP	24	LCP array in text order
$\mathsf{PSV}_{\mathsf{lex}}$	34	PSV array in lexicographical order
$\mathsf{PSV}_{\mathsf{text}}$	35	PSV array in text order
$\mathcal{S}$	-	Stack
SA	7	Suffix array
$\mathsf{SA}_{S:S'}$	12	SA of string SS' restricted to suffixes starting in S
T	7	Input text
TiB	-	Tebibyte (2^{40} bytes)
W	40	Concatenation of text segments to the left of X
X	13	Currently processed text segment
Y	13	Concatenation of text segments to the right of X
c	-	Character
d	13	Number of text segments
f_i	8	i^{th} factor in the LZ77 parsing
$\mathsf{gap}_{S:S'}$	13	Gap array
i, j, k, ℓ	-	Non-negative integers
$\mathsf{lcp}(S, S')$	8	Length of the longest common prefix of strings S and S'

Symbol	Page	Semantics
ℓ_i	8	The length-component of $\mathsf{LPF}[i]$
$\widehat{\ell}_i$	40	The length-component of $\mathsf{MS}[i]$
m	13	Length of segment of text
n	7	Length of text
n_i	30	Length of text in the i^{th} round of partial processing
p_i	8	The position-component of $\mathsf{LPF}[i]$
$\widehat{p}_i$	40	The position-component of $\mathsf{MS}[i]$
q	30	Number of partial processing rounds
r	39	Number of runs in BWT
$\mathsf{rank}_c(S, i)$	16	Number of occurrences of character c in $S[0..i)$
t_{rank}	40	Time complexity of a rank query
z	8	Number of phrases in the LZ77 parsing
Σ	7	Alphabet
Φ	24	An array such that $\Phi[\mathsf{SA}[i]] = \mathsf{SA}[i-1]$
σ	7	Alphabet size